LEHRSTUHL FÜR REALZEIT-COMPUTERSYSTEME
TECHNISCHE UNIVERSITÄT MÜNCHEN
UNIV.-PROF. DR.-ING. G. FÄRBER

Design and Implementation of an FPGA-based Image Processing Framework for the EyeBot M6

Martin Geier

Diplomarbeit

Design and Implementation of an FPGA-based Image Processing Framework for the EyeBot M6

Diplomarbeit

Supervised by the Institute for Real-Time Computer Systems
Technische Universität München
Prof. Dr.-Ing. Georg Färber

Executed at the Centre for Intelligent Information Processing Systems
University of Western Australia
Perth, Australia

Advisor: Dipl.–Ing. Sebastian Drössler

Author: Martin Geier
mgeier@mytum.de

Submitted in February 2009

Abstract

The EyeBot M6 is the newest revision of an embedded system designated for the control of small mobile robots. Unlike previous revisions of the system, the EyeBot M6 features not only a 400 MHz CPU running a fully fledged operating system but also a Xilinx FPGA accompanied by an SRAM and two cameras in a stereo setup. The recent advancements in FPGA fabrication not only induced lower prices but also permit the implementation of large signal processing algorithms in FPGAs. The current revision is the first EyeBot that tries to exploit the increasing capabilities of FPGAs for image processing purposes on small robots.

This project focuses both on the low-level interfacing between the FPGA and the CPU and on the internal memory bus architecture required for image processing purposes.

Previous work on the EyeBot M6 already showed that the communication between CPU and FPGA was unreliable. Sporadic transfer errors (mostly occurring during long DMA transfers, after FPGA design modifications or toolchain configuration changes) had a severe impact on reliability and maintainability of the system.
In this thesis first of all an in-depth analysis of the previous system is undertaken and reveals several potential sources of error. The old system disregards the fact that CPU and FPGA are in distinct clock domains and therefore is prone to timing violations. Timing violations may occur in a register if one of its input signals changes too close to the clock edge that triggers the register. Because CPU and FPGA clocks are fully unrelated, a signal generated by the CPU may change close to an FPGA clock edge. In consequence this may induce a timing violation in a register inside the FPGA that samples this signal. The same scenario may happen in reverse, too.
Timing violations may lead to a metastable state in the particular register. Because this results in unpredictable behaviour of the register's output the proper operation of all downstream units is compromised.
In addition, it is found that an important step in the FPGA design flow has been omitted: Timing constraints are required to inform the development toolchain of the timing requirements caused by the components connected to the FPGA. If no constraints are applied to an FPGA design the toolchain will only implement the design according to its internal objectives. This might lead to an FPGA design that violates the timing requirements of the external components and thus to transfer errors, too.

As a next step, possible solutions for the problem related to register timing violations are investigated. Several straightforward approaches (that would convert the design into a fully synchronous system) are found but can not be deployed on the EyeBot M6. The asynchronous interface therefore has to be utilized and used to interconnect CPU and FPGA in a safe manner. Because of its asynchronous nature timing violations can not be fully avoided and their possible

outcome therefore is evaluated. The probability of metastable events affecting the operation can be minimized using the gained knowledge on metastability. Based on these findings two interfacing approaches are identified and assessed.
One interface supports arbitrary accesses to storage locations inside the FPGA but is unable to achieve the highest possible transfer rate on the bus between CPU and FPGA. The other interface, however, is capable of fully loading the bus but comes at the cost of setup-overhead. The former therefore is suitable for transmitting small amounts of data (e.g. for measurement or control purposes) and the latter is adequate for large, continuous transfers (e.g. images).
The interfacing requirements are analyzed based on the expected data flow between CPU and FPGA. Based thereon, both interfaces are considered necessary and thus selected for implementation. The two interfaces are integrated into a bus bridge that terminates the asynchronous VLIO bus and instantiates a fully synchronous internal bus architecture.
The operativeness of the bus bridge is verified both using simulation and testing on the actual hardware. Simulation alone is considered insufficient because the behaviour of a metastable register can not be modelled.

As a next step, timing constraints are applied to the new design. Regular synchronous constraints are used to specify the timing requirements of the CPU and the two cameras connected to the FPGA. In addition, several combinatorial constraints are applied to cover the asynchronous paths in the bus bridge.

In a final step, a storage architecture required to access the external SRAM is designed. External storage is required by an image processing system that was developed simultaneously by a fellow student (see [29] for reference). The image processing system utilizes the stereo cameras of the EyeBot M6 and pursues the generation of a depth map.
Its requirements are analyzed and the required high-level flow control is determined. Based thereon, the storage architecture is tailored to the estimated data flows generated by the image processing system. The existence of several clock domains in the system rules out a static, predefined arbitration scheme. Instead a simple priority-based arbitration is used to determine which flow is handled next by the storage system. Based on a performance estimation the achievable maximum frame rate of the complete system is given.
However, it is found impossible to implement the proposed storage architecture with the current revision of the PCB. Due to a missing external clock feedback trace the clock skew between the FPGA and the external SRAM can not be compensated. This rules out the application of timing constraints for the SRAM as well.
Clock skew is a common problem in synchronous systems and Xilinx FPGAs contain dedicated resources capable of compensating it. A solution deploying two phase locked loops is found and the modifications required on the PCB are shown.

Further suggestions for improvement of the platform are presented in the appendix.

Acknowledgements

I would like to thank A/Prof. Dr. Thomas Bräunl for offering the possibility to work on this project here in Perth.

I also would like to thank Prof. Dr. Georg Färber and Dipl. Ing. Sebastian Drössler for their support and guidance during this thesis.

Many thanks go to Benedikt Dietrich for the great teamwork during all the time. Thanks to my fellow group members Azman M. Yusof and Soo Teoh for their ideas and support.

Thanks also go to Beni, Vinnie, Markus, Steven, Jenny, Owen and all others for a great time in Perth!

Finally, I would like to thank my family and all my friends at home for their sympathy and support during my time abroad.

Contents

List of Figures

List of Tables

List of Symbols

BB	*Burst Bus*: See sections 4.2.6 and 4.4.1 for more information.
CD	*Clock Domain*: A clock domain is defined as a group of synchronous elements (registers, BRAMs, multipliers, etc.) that are driven by one and the same clock source (or by multiple clocks with fixed phase relationship). Signals generated in this clock domain will change synchronously, that is after the active edge of the clock. The minimum clock period of the circuit can be determined by identifying the delay of the longest path between two synchronous elements and adding it to the register's setup- and clock-to-output-times. By driving the circuit with a frequency that adheres to this minimum clock period it can be assured that no timing-violations will occur.
CDC	*Clock Domain Crossing*: A clock domain crossing results from a signal being generated in one CD and being sampled in another CD. Setup- and hold-violations in the receiving register can not be avoided because the phase relation between the two clocks is unknown.
CLB	*Configurable Logic Block*
CPU	*Central Processing Unit*
DCM	*Digital Clock Manager*: A component inside a Xilinx FPGA used to generate internal and external clocks. It is capable of eliminating skew and synthesizing new clock signals and frequencies. Skew elimination on a net CLK driven by the DCM generally works by measuring the phase difference between the DCM's clock input (CLKIN) and the skewed version of the driven clock net (CLKFB) which therefore has to be fed back into the DCM. The DCM deploys a closed control loop that phase-shifts CLK in such a way that CLKIN and CLKFB are in phase which eliminates the skew on CLK. The internal clock nets of the FPGA can be fed back to the DCM using the FPGA's internal routing. All skew-sensitive external clocks driven by the FPGA however have to be fed back manually by the PCB to allow for the DCM-based skew compensation.
DMA	*Direct Memory Access*
FIFO	*First In First Out*
FPGA	*Field-Programmable Gate Array*

GCLK	*Global Clock (input resp. net)*
JTAG	*Joint Test Action Group*
LUT	*Lookup Table*
MMIO	*Memory-Mapped I/O*
MTBF	*Mean Time Between Failures*
PB	*Peripheral Bus*: See sections 4.2.6 and 4.2.7.2 for more information.
PCB	*Printed Circuit Board*
PSD	*Position Sensitive Device*
PWM	*Pulse-Width Modulation*
RCS	*Revision Control System*: A software used to track changes in source files and to support multiple developers in exchanging and merging their contributions to a project. Common revision control systems are subversion, BitKeeper and ClearCase.
RTL	*Register Transfer Level*
SDRAM	*Synchronous Dynamic Random Access Memory*
SRAM	*Static Random Access Memory*
TPU	*Time Processor Unit*: The TPU is a coprocessor included in certain Motorola 683xx processors. It supports various timer functions, e.g. capture and compare or PWM generation.
TT	*Transfer Type*: See section 4.4.1.4 for more information.
USB	*Universal Serial Bus*
VHDL	*VHSIC Hardware Description Language*
VHSIC	*Very-High-Speed Integrated Circuits*
VLIO	*Variable Latency I/O*: VLIO is a bus interface mode of the Intel PXA255 processor's memory controller. It allows an external device to lengthen a bus cycle on demand.

1 Introduction

1.1 Context of this thesis

This thesis is associated with the ongoing development of a new computing platform for mobile robots at the Centre for Intelligent Information Processing Systems of the University of Western Australia. This platform, the EyeBot M6, was created to supersede previous versions of that controller in terms of speed and to allow research on hardware-accelerated image processing algorithms. As previous versions it features an LCD, several communication ports, various sensor inputs and actuator outputs as well as a camera interface. It is comprised of a Linux-based embedded system (which even provides Ethernet and Bluetooth connectivity) and a Xilinx Spartan-3E FPGA, all on one small PCB.

The inclusion of an FPGA is the major advancement compared to older models, because it allows the deployment of hardware/software co-design techniques to distribute the various computation-tasks among soft- and hardware. This enables the user to implement more complex algorithms, mainly in terms of image processing. Besides that, the EyeBot M6 is the first EyeBot that actually runs a fully fledged operating system (Linux), which implicates a different approach in writing software.
A few peculiarities and differences to other systems have to be noted firstly:

- The EyeBot M6 was (and is) developed, manufactured and tested by students and employees of the School. This allowed the system to be contrived to match a lot of different requirements (e.g. size, cost or power consumption).
- As a result of the long development process (which started in 2006), the components used to build the EyeBot M6 now may seem to be a little outdated. This is particularly true for the Spartan-3E FPGA, which has been overhauled by newer devices in terms of logic count, size and architectural features. Nonetheless, the system forms a solid base for research, provided that the limitations imposed by the FPGA are taken into consideration when assessing the performance of the system.
- Because of the EyeBot M6 being a custom design, various hard- and software-building blocks needed to form the system had (and still have) to be developed from scratch. This not only includes software components (e.g. I/O-libraries) but also all hardware modules that are required in the FPGA. Bringing the EyeBot M6-platform into an usable state is quite a complex task compared to other similar, commercial platforms featuring a CPU and an FPGA (such as the Xilinx Virtex-5 FPGA). The reason therefore is that even basic capabilities such as data transfers between CPU

and FPGA have to be implemented by hand. This includes the development of drivers (on the software side) and of appropriate HDL-modules (on the hardware side) to interface to the CPU. Commercial CPU-FPGA-platforms normally not only provide this basic features/capabilities but also include various high level modules by being shipped with a comprehensive library of software and IP-cores.

Previous work on the EyeBot M6 had already shown that the data communication between CPU and FPGA is unreliable. The reasons for it were unknown at that time.

1.2 Addressed tasks

This project focuses on the low-level building blocks required to use the EyeBot M6 for image processing and robotics purposes.
The primary goal is to establish a reliable link-up between CPU and FPGA. This comprises an in-depth analysis of the failing system at hand, an extensive research in interfacing approaches and associated topics and the actual problem solving. The resulting system has to be verified carefully and the achievable performance of the interface has to be evaluated, too.
The secondary goal is the development of an internal storage architecture for the FPGA. An external SRAM is used to extend the limited internal memory of the FPGA and is required by the image processing system designated for the EyeBot M6. Its requirements have to be analyzed and taken into consideration when designing the storage architecture.

1.3 Outline of this thesis

Chapter 2 overviews the platform used during this project. The EyeBot M6, its origin, evolution and features are presented. Existing hard- and software is covered, too.

Chapter 3 takes a closer look at the hardware and the existing VHDL design. Based thereon, two sources of the observed transfer errors are identified. The first is caused by the existence of several clock domains in the system while the second is a result of an omitted step in the FPGA design flow. These errors are also found to be responsible for a problem related to the cameras and the external SRAM.

Chapter 4 presents all steps of the development and begins with a short overview.
Section 4.2 describes the bus bridge which links CPU and FPGA in a reliable way. The chosen implementation solves the first source of error identified in chapter 3.
Section 4.3 catches up on the omitted step in the FPGA design flow and introduces timing constraints. This solves the second source of error identified previously.
Section 4.4 presents the storage architecture developed for the image processing system that is designated for the EyeBot M6 (and pursues the generation of a depth map).

Chapter 5 summarizes both findings and results of this project and suggests future work.

2 The EyeBot M6

2.1 Introduction

The platform used in this thesis is the EyeBot M6, mainly described in [25]. It is the newest of all embedded systems developed at the Robotics and Automation Lab and seeks to be compatible with the established API called RoBIOS (more information can be found in [9]). This library provides the user with various functions to access all hardware needed to control a robot and also provides an interface to OS-like features (e.g. multitasking, timers, semaphores, etc.). Additionally, a simulator software is available which allows testing of programs depending on RoBIOS even without having an actual robot at hand. In the following a short overview of the EyeBot M6 will be given.

The EyeBot M6 was intended to supersede previous versions of the EyeBot-controller. These old EyeBots were based on a Motorola 68332-CPU and allowed the connection of one small CMOS-camera for image acquisition and processing purposes. Because of the limited processing power of the CPU a camera with a resolution of only 80×60 px was used. Furthermore, various I/O devices were available, such as an LCD, some buttons, serial ports, motor and servo drivers, encoder and PSD inputs, general purpose I/O lines, infrared and audio.

The new EyeBot was supposed to have at least the same number of interfaces for reasons of backward compatibility. Additionally, some new ones were required, for example USB (to cope with the evolving lack of serial ports on modern computers), Ethernet (for IP networking) and a second camera port (to allow research on stereo vision). Three major changes were done to account for this increased demand for processing power and additional communication methods:

- The Motorola 68332 got replaced by an Intel XScale PXA255,
- the RoBIOS operating system got replaced by a fully fledged Linux kernel and
- the CPU was accompanied by an FPGA.

This changes had (and still have) huge impact on all the development related to the new revision of the EyeBot:

- A new development toolchain was introduced
- All RoBIOS-OS-related functions had to be ported to their Linux equivalent

- All previously CPU-supported functions[1)] have to be replaced either by software or by suitable hardware modules (dedicated ICs or logic inside the FPGA)
- Kernel drivers have to be developed to gain access to all these modules and all RoBIOS-library functions have to be modified
- Documentation describing the new architecture has to be written to support new developers in getting used to the system
- The RoBIOS-API documentation has to be updated to reflect the changes visible to the user interface

At the time of this writing most of these tasks are still in progress or have not been started yet. Figure 2.1 shows the EyeBot M6 controller mounted onto a robot.

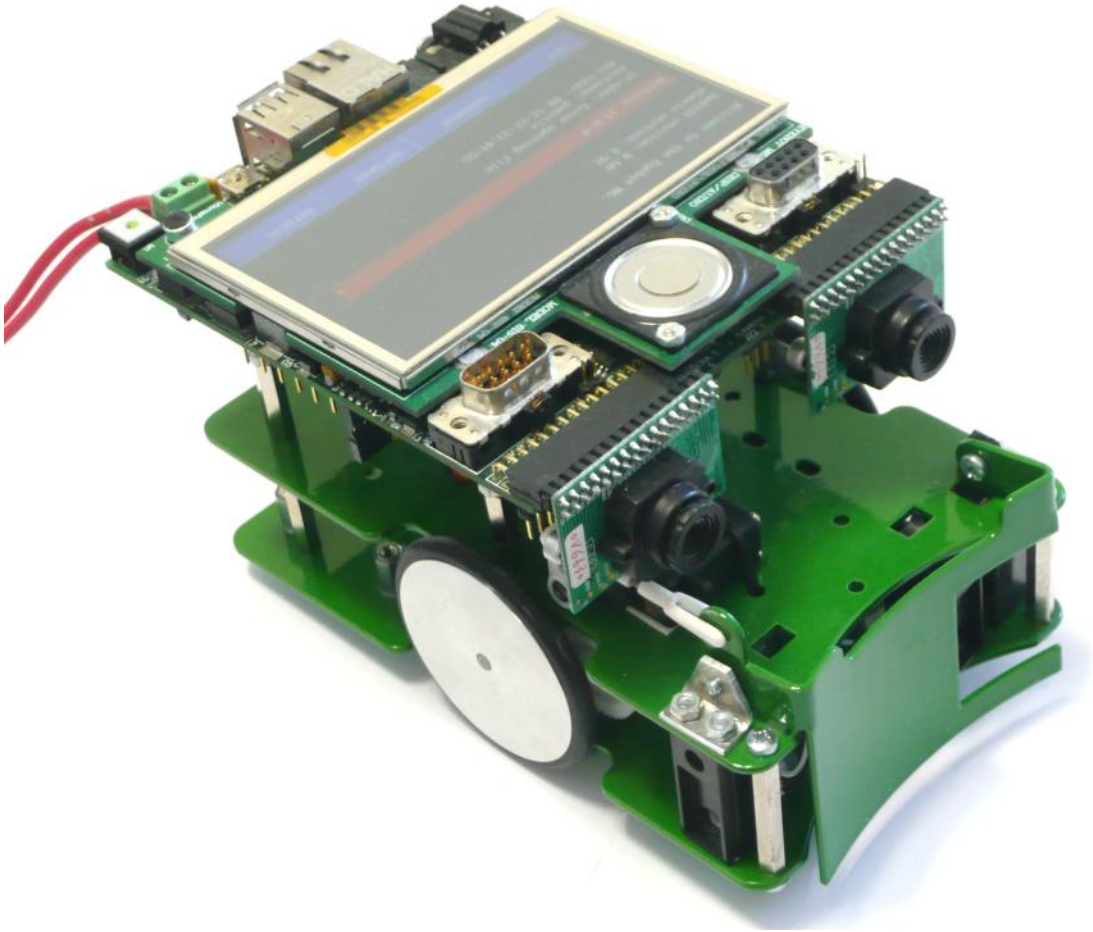

Figure 2.1: The EyeBot M6

The next two paragraphs introduce the parts of hard- and software that are related to this thesis. A more detailed description is given in [25].

[1)] such as PWM generation that had been handled by the TPU before

2.2 Hardware

During the development of the EyeBot M6 some importance was attached to avoiding ICs in BGA packages (to enable the Electronic Workshop of the School to populate the PCB). This decision ruled out a lot of fast, up-to-date CPUs, what seemed inappropriate for the next generation of EyeBots. Thus, an alternative approach was taken: Instead of a CPU a single board computer (connex 400xm-bt) from a company named gumstix (see [3]) was purchased and used as foundation for the new EyeBot. It features a 400 MHz Intel XScale PXA255-CPU, 64 MB RAM, 16 MB Flash, a Bluetooth module and two connectors for external components.

The external bus interface of the PXA-CPU is exposed on one of these connectors and uses a transfer protocol named VLIO (see chapter 6 of [5]). This bus was used to create a connection between the CPU and various components, such as an Ethernet controller, an USB-host controller and the FPGA.

Again for packaging reasons a medium-sized FPGA was selected, namely the Xilinx Spartan-3E 500 (XC3S500E PG208). It features 4656 Slices (each containing two 16 × 1 bit-LUTs, two 1 bit registers and more), 360 kBit of internal (dual-ported) SRAM (so-called "Block-RAM") and 20 multipliers. The pin-count of the package limits the device to 158 I/O-Pins (single-ended), which also limited the connection to the CPU data bus to a width of 16. A more detailed look on the interface between CPU and FPGA is given in section 3.3.

The FPGA is connected to various other components, such as

- PSD and encoder inputs,
- motor drivers,
- servo outputs,
- two camera ports,
- a 50 MHz oscillator and
- a 18 MBit (single-ported) SRAM.

Figure 2.2 shows a coarse block diagram of the EyeBot M6.

The used cameras are capable of capturing at a resolution of 352 × 288 px and support various output modes (e.g. RGB and YUV). An image of this size consists of 99 kPx, so a grayscale image (with 8 bit per px) will have a size of 99 kB or 792 kBit – too much for the internal SRAM of the FPGA. Thus, if an entire camera frame shall be stored, the external SRAM has to be utilized.

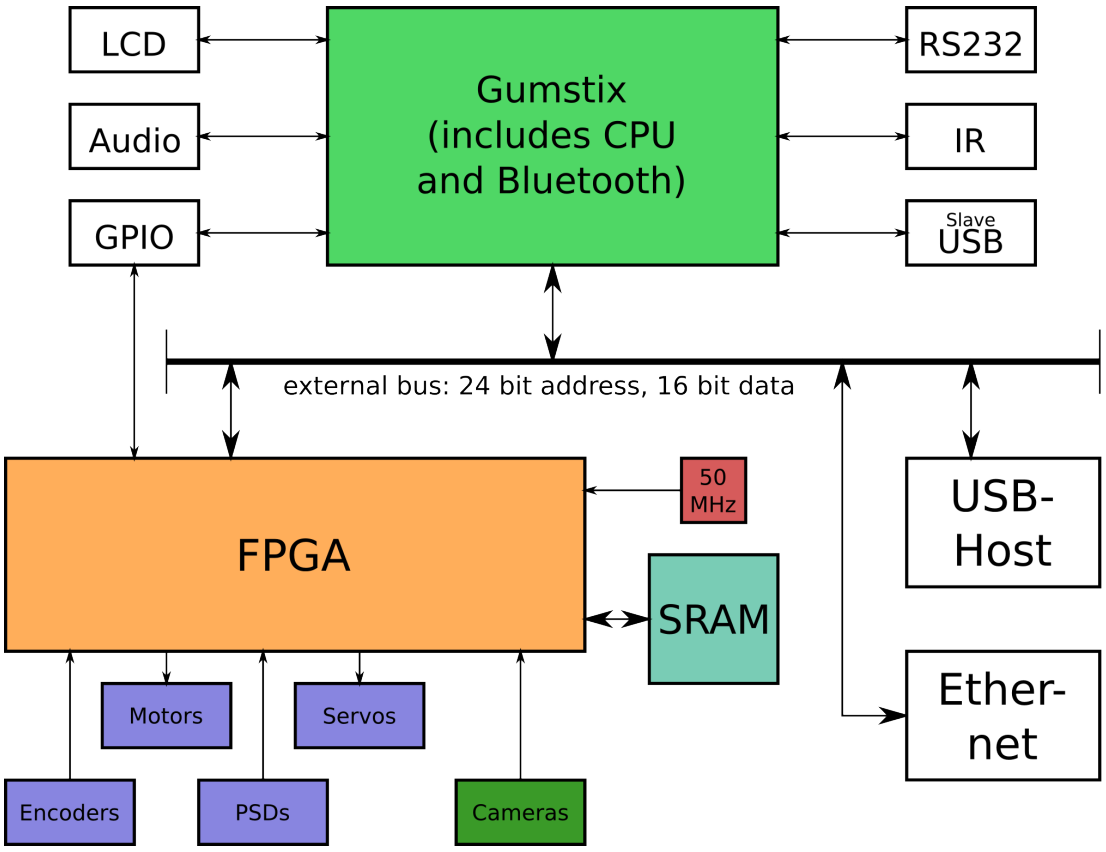

Figure 2.2: Block diagram of the EyeBot M6

2.3 Software

The gumstix connex is shipped with a pre-installed version of the Linux operating system on its flash memory. To enable the user to modify and upgrade the system, gumstix extended a software package called buildroot (see [1]) to support their hardware. Thereto, various patches (for kernel and user space) were added and made available via SVN. In combination with freely available software (as e.g. GCC or the Linux kernel) the buildroot package forms the complete build environment needed to recreate the root file system residing on the flash memory.
The kernel patches from gumstix did (obviously) not include support for Ethernet controller, USB-host and the FPGA. Drivers for Ethernet- and USB-controller were created by adapting drivers of similar chips that already were included in the Linux kernel.
A driver for programming and transferring data from and to the FPGA was written from scratch. It supports both classic MMIO- and DMA-based transfers including DMA-to-userspace which results in higher transfer rates.
To provide a similar "look and feel" as previous versions of the EyeBot, a software called m6main was written. It initializes the LCD, displays various pieces of information (name of the system, IP address, etc.) and enables the user to start his/her programs by selecting them in a graphical file browser.

3 Analysis of the existing platform

3.1 Introduction

An FPGA design for the EyeBot M6 was already under way at the beginning of this thesis. Additionally, some software libraries for accessing these modules have been developed, trying to imitate the known RoBIOS-interface (which has been provided by previous versions of the EyeBot).

The following modules did already exist:

- a bus interface (simply routing the external VLIO to an internal bus)
- an address-decoding unit (producing various enable signals)
- an SRAM interface (allowing access from both CPU and an internal camera module)
- various peripheral modules for digital I/O, motor control (PWM and encoder-acquisition), servo driving (PWM), PSD measurement acquisition (serial-to-parallel conversion) and camera frame-grabbing (including image storage to SRAM)

A serious problem had already been noticed when the author of this thesis joined the development team: The communication between CPU and FPGA tended towards being unreliable (in terms of sporadic data corruption) – for (at that time) unknown reasons. The problem could by triggered by

1. executing DMA transfers with 128 or more beats or by
2. applying certain modifications (e.g. rearranging registers or extending the design).

Either scenario depicts a serious issue, the former because it renders fast transfers of data impossible, the latter because it inhibited the development process.

A thorough analysis of (and the explanation for) this problem is given in sections 3.3 and 3.4. The problem solving formed a significant part of this thesis and is described in section 4.2.

Apart from the transfer errors between CPU and FPGA another flaw has been found: When streaming video from the cameras noise-like pixel errors appeared in the stream. The reason was found to be similar to the one generating the bus transfer errors above. More information on this topic can be found in section 3.5.

3.2 General notes

3.2.1 VHDL

The VHDL modules were mostly written in behavioural VHDL, though they were never intended (and used) for simulation. This led to various problems for everyone trying to explore the design using simulation, e.g. runtime errors because of range exceptions or simulation mismatches because of missing initialisation values.

Even tough simulation tools (such as Modelsim) support a much larger subset of VHDL compared to synthesis tools (as XST), it is still inevitable to keep an appropriate coding-style at the back of one's mind. (But for obvious reasons the capabilities of the synthesis tool also have to be considered.)

A coding style that accounts for both comprises various factors:

- Initialize register values (to avoid uninitialized bits (`'U'`) in simulation)
- Use `std_logic(_vector)` (at least for ports)
- Be very careful when using resets (especially when using asynchronous ones)
- Take care that integer-based types do not overflow in simulation
- Avoid latches (use registers as they simplify timing analysis)
- Keep sensitivity lists up-to-date
- Use `rising_edge(foo)` instead of `foo'EVENT and foo='1'`[1)]
- Don't use signals for storage (which would decrease simulation speed)
- Avoid complex operations as e.g. division if possible (saves logic resources)
- Do not use wait `for` (delay lines are rarely available in FPGAs)

Finally, a designer keeping both simulation and synthesis in mind (while writing HDL code) will be rewarded by a simulation that matches hardware behaviour (as far as possible) and therefore alleviates debugging.

Further suggestions can be found in in [23] and in [24].

3.2.2 Documentation and revision control

Also some infrastructural issues were encountered:

- The previous FPGA design (introduced in section 3.1) was sparsely documented.

[1)] `rising_edge()` also detects transitions from e.g. `'L'` to `'H'`, which can occur in simulation. This is important because XST will map `'L'` to `'0'` and `'H'` to `'1'`.

- Some former theses included VHDL code designated to the EyeBot M6, but at least one of them already had been decided unusable (because of the complete lack of documentation).
- The various versions of the source code (of the FPGA design) were not stored in an RCS. This hampered comparison between working and non-working designs which may have eased finding the reason for the transfer errors.
- The source code of the modified RoBIOS library was neither stored at a central location nor maintained in an RCS, even tough multiple developers were updating the library at the same time.
- The DMA routines of the kernel driver for accessing the FPGA have not been documented[2)].

All these factors hindered new developers in joining the project.

3.3 Interface between FPGA and CPU

In the first part of this section the electrical interface between CPU and FPGA is described and the challenges for a safe data transmission are identified. After that, the previous FPGA design (henceforth called "old design") is studied and further flaws are demonstrated. Finally, the resulting set of problems is evolved from the previous findings.

3.3.1 Electrical interface and bus protocol

This section firstly introduces the memory interface of the PXA255-CPU and its various modes of operation. After that, it focuses on the existing electrical interface between CPU and FPGA and derives useable transfer modes.

The memory controller of the PXA255 processor supports various types of memories, categorized into three groups according to their interface:

Synchronous dynamic RAM (SDRAM): The most important property of SDRAM is its need for periodic refreshes to retain the saved data. Up to four SDRAM chips (16 bit or 32 bit wide) with a maximal size of 64 MB each are supported. All signals are synchronized to a clock signal.

Synchronous static memory: Static memories, as e.g. SRAM (volatile) or Flash (non-volatile), retain their data without a continuous refresh. As with SDRAM all signals are synchronized to a clock signal from the memory controller, which supports up to four of these devices.

[2)] This in particular was a serious problem because the author of the driver deployed a tricky method to store the buffer address, which is needed to start a DMA-to-userspace transfer. Someone not being aware of that has great difficulties to reconstruct the driver's mode of operation.

Asynchronous static memory: Asynchronous memories however do not make use of a clock signal at all. Reads are performed by applying an address, enabling the memory and reading the data after a device-specific delay. For writing address and data are supplied first and write-enable is asserted. After a certain time write-enable is deasserted again. A special mode called "variable latency I/O" (VLIO) enables external devices to lengthen the bus cycle by adding CPU wait states. This is useful for devices with variable response time.

Each mode make use of a specific (sub-)set of control-pins of the memory controller. Table 3.1 describes the most important signals. For space reasons "external device" is written as "ED".

Pin	Name	Direction	Comment
MA[25:0]	address bus	CPU → ED	26 bit can address up to 64 MB
MD[31:0]	data bus	bidirectional	can be configured to a 16 bit data bus
$\overline{\text{OE}}$	output enable	CPU → ED	notifies the ED to drive the data bus
$\overline{\text{WE}}$	write enable	CPU → ED	notifies the ED that valid data is on the bus
$\overline{\text{PWE}}$	PCMCIA $\overline{\text{WE}}$	CPU → ED	same as $\overline{\text{WE}}$, but used for PCMCIA and VLIO
$\overline{\text{SDCS[3:0]}}$	SDRAM select	CPU → ED	4 chip selects (one for each SDRAM chip)
SDCLK[2:1]	SRAM clock	CPU → ED	2 clock signals (sync. static interface)
$\overline{\text{CS[5:0]}}$	SRAM select	CPU → ED	6 chip selects (sync. and async. static interface)
RD/$\overline{\text{WR}}$	direction	CPU → ED	identifies the direction of the transfer
RDY	device ready	CPU ← ED	only used with VLIO

Table 3.1: Ports of the PXA255's memory controller

Figure 3.1 presents a more detailed view on the circuitry on the EyeBot M6. Compared to figure 2.2 some unimportant components have been omitted. Please note that (due to the lack of a schematic from gumstix) the connections shown onboard of the gumstix may not reflect the actual circuitry.

At first sight a shared bus between CPU, flash, SDRAM, FPGA, Ethernet- and USB-host-controller can be identified. This imposes a first limit on the maximal data rate.

On closer inspection, more details can be found:

- The RDY-pin is only connected in later versions of the EyeBot M6
- Data and address lines are routed through bidirectional transceivers (buffers)
- Transfer qualifiers such as RDY, $\overline{\text{OE}}$ or $\overline{\text{PWE}}$ are not buffered
- RD/$\overline{\text{WR}}$ is not connected to the FPGA
- There is no clock signal leaving the gumstix

Because of various reasons (given in section 4.2.1) the only usable transfer mode is VLIO. Therefore, only VLIO will be presented. Figure 3.2 shows a write access, figure 3.3 a read

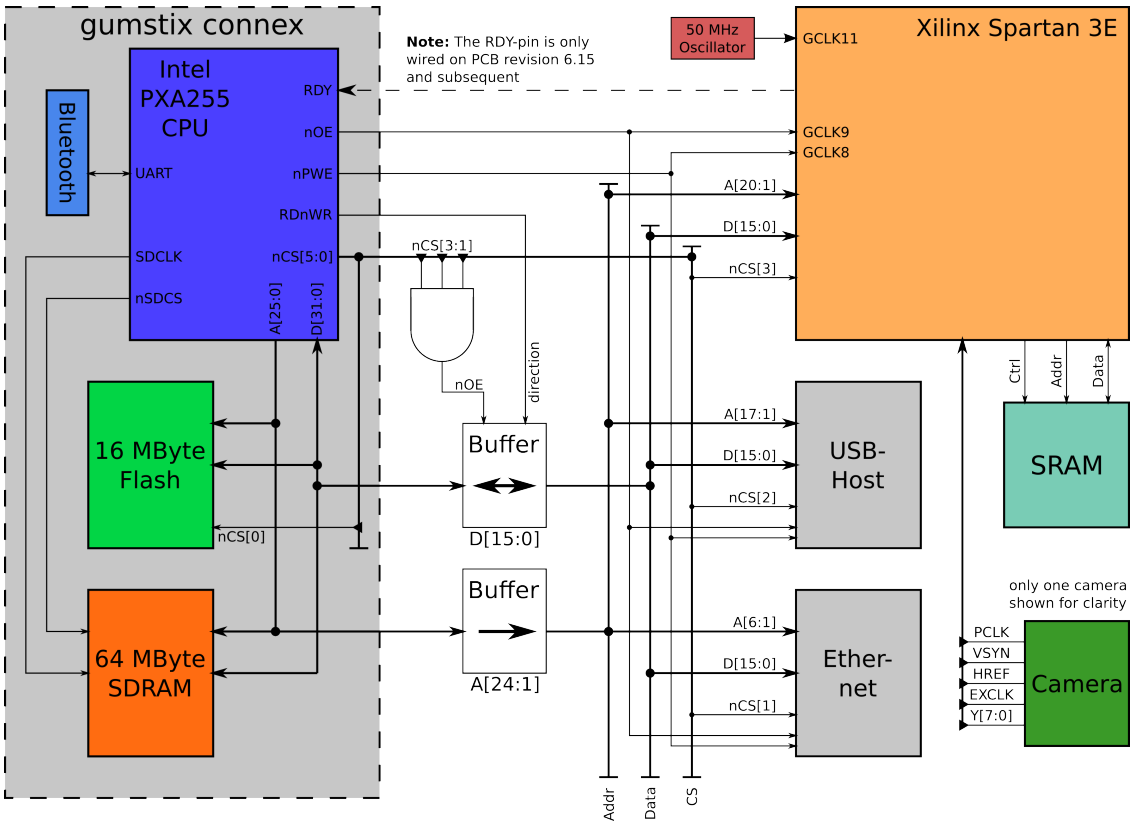

Figure 3.1: Schematic view of FPGA-CPU-interface

access using the VLIO protocol in non-burst mode. (Burst mode will be introduced in section 4.2.5.2.)
Timing values are given as multiple of MEMCLK = 10 ns (f_{MEMCLK} = 100 MHz).

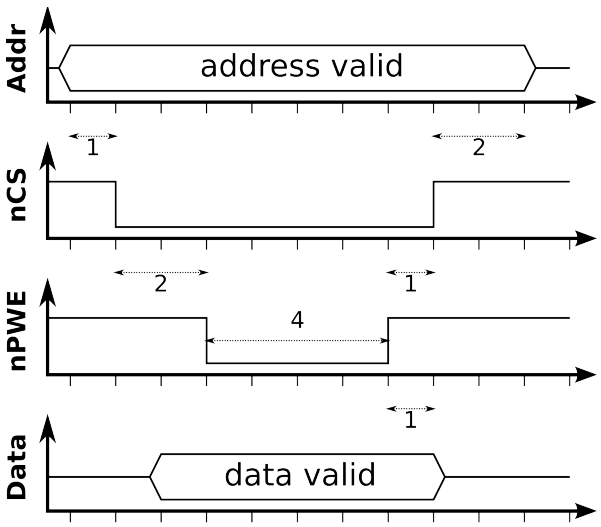

Figure 3.2: VLIO write

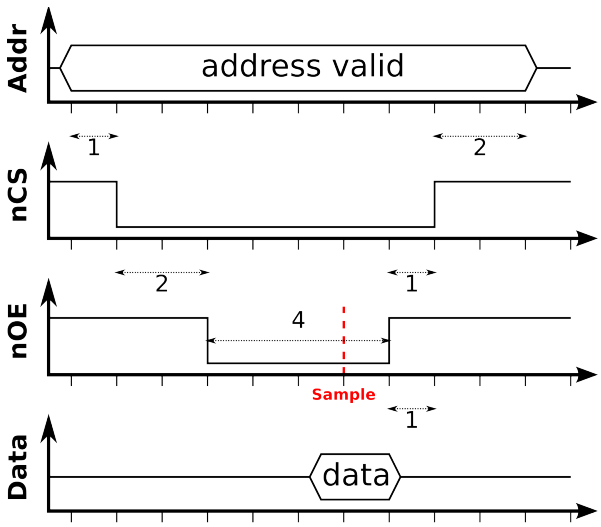

Figure 3.3: VLIO read

The sequence of signal changes is mostly the same for write and read: First and foremost, an address is driven to the address bus and $\overline{\text{CS}}$ gets asserted[3)] one clock later. After two further clocks $\overline{\text{OE}}$ or $\overline{\text{PWE}}$ gets asserted for (at least) 4 clocks, depending on the state of the RDY-pin (which will be described in section 4.2.5.1). For this first introduction on VLIO the RDY-pin will be assumed to be high at all times.

On a write access the data bus has been valid since one clock before the assertion and will be valid until one clock after the deassertion of $\overline{\text{PWE}}$. An external device being written to has to latch data and address during that time.

On a read access the data bus will be sampled by the CPU 3 clocks after the assertion of $\overline{\text{OE}}$ and has to be stable at least 1.5 clocks before the deassertion of $\overline{\text{OE}}$. Thus, the external device has to drive the data bus to a valid value in 2.5 clocks (or less) after the assertion of $\overline{\text{OE}}$. Considering the delay of the buffer on the data bus ($< 5.2\,\text{ns}$), less than 20 ns are available to the external device to answer the read request.

Because of inconsistent values in the data sheets of the PXA255 a table containing all known timing parameters is included in appendix A.

3.3.2 Analysis of the existing FPGA design

Initially the author started exploring the existing FPGA design to familiarize himself with the partitioning and the used coding style. During that process some design flaws have been noticed. These flaws and their impact on the system will be described in the following.

Figure 3.4 shows a cut-out of the existing design. For the sake of clarity the circuitry is drawn on RTL even though it was extracted from the actual implementation on the FPGA (by analyzing CLB- and routing-information). The I/O-pins connecting FPGA and CPU are shown on the left hand side and are prefixed with `CPU_`. The link-up of the various internal modules to the CPU is exemplified with a single 16 bit register (`FD`) of the `xdio`-module. The module itself handles digital I/O and uses the same internal architecture as all other modules.

First and foremost, three modules can be identified: The leftmost (`cpu_interface`) generates two control signals (`cpu_nrd_flag` and `cpu_nwr_flag`) and splits the incoming address into two parts. The next module (`mod_interface`) then uses these signals to decode a part of the address and to drive both the transfer qualifiers (`mod_rqrd` and `mod_rqwr`) and a sub-address bus of the internal module bus. Among others the `xdio`-module is connected to that bus. It contains a 16 bit register that is used to configure, read and set each single I/O-line. This functionality is not illustrated in figure 3.4.

On closer examination an issue concerning the timing of the two registers can be identified. Both registers are clocked by internal clocks derived from the clock-input `GCLK11`. The latter is connected to an external stand-alone 50 MHz oscillator on the EyeBot-PCB (see

[3)] An inverted signal is considered asserted when it is driven low.

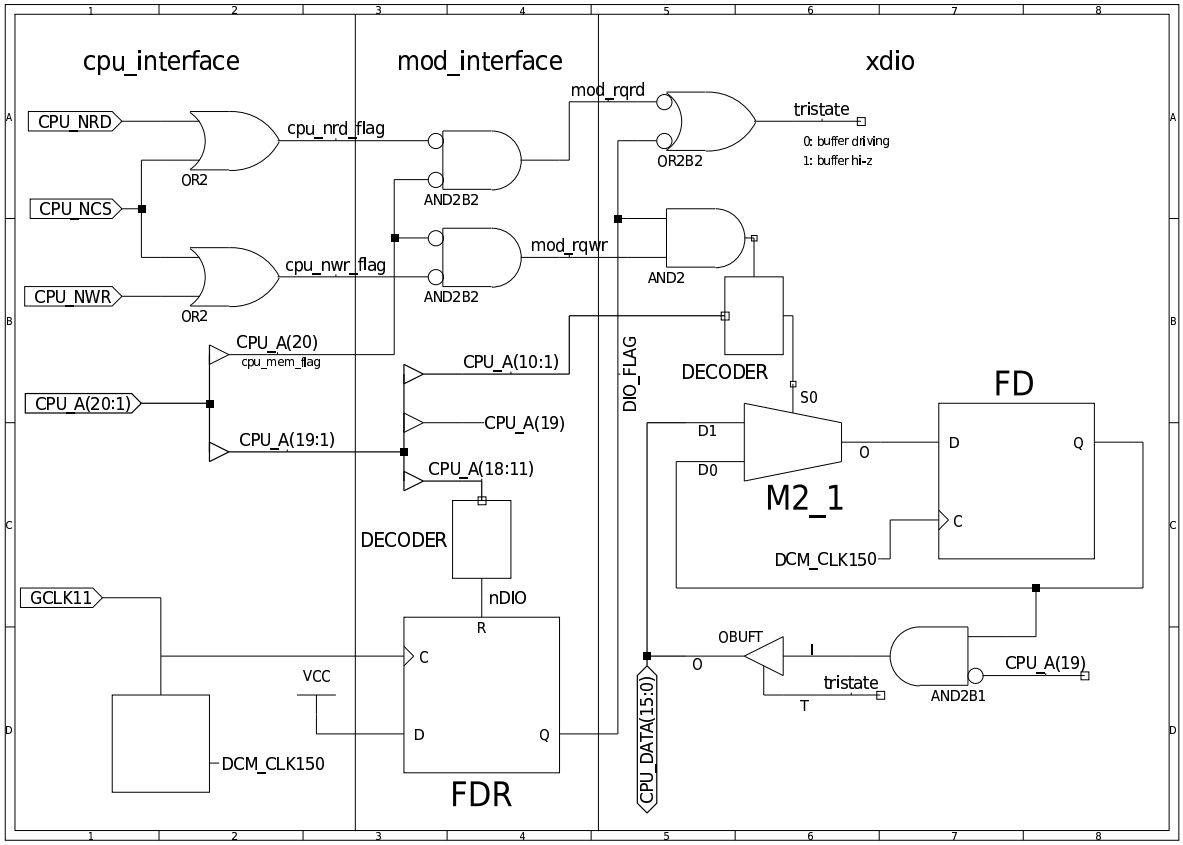

Figure 3.4: Old FPGA design (cut-out only)

figure 3.1). From this follows that both internal clocks have to be considered asynchronous to all other clocks on the EyeBot M6. This is particularly true for all clocks of the PXA-CPU (which are derived from a crystal on the gumstix-PCB) including `MEMCLK` indicated in figure 3.2 and figure 3.3.

Before continuing the derivation a short introduction to registers and their timing-requirements will be given.

Registers and setup- and hold-time-violations

A register is the basic element used to store a value in a synchronous digital circuit. In a synchronous FPGA design a register can be treated as a "black box" that copies the value of its D-input to the Q-output on every rising edge on the clock-input.

Because of its physical nature a register applies certain requirements on the timing of the signals. These constraints will be exemplified by the waveforms given in figure 3.5.

First of all, a register obviously only supports clock frequencies up to a specific value (which defines a lower bound for the clock period). Additionally,

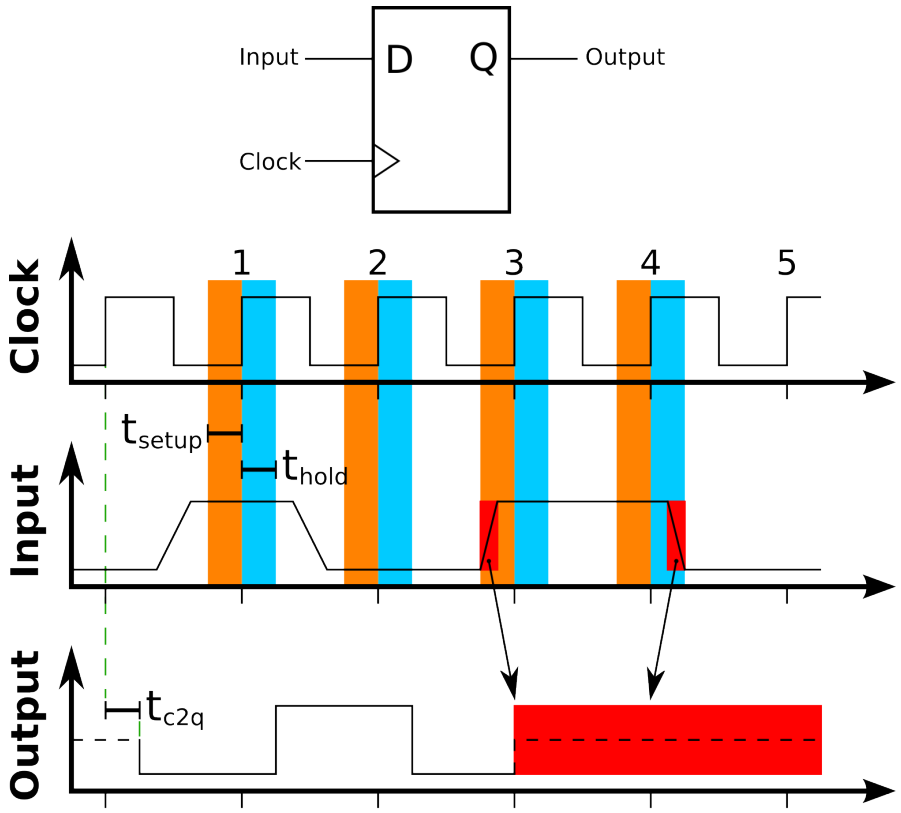

Figure 3.5: Register timing

the so called "clock to output" time (t_{c2q}[4]) gives the maximum delay for data (copied from the input) showing up at the output after the active clock edge. Finally, there are two important restrictions related to changes of the input signal happening close to the active clock edge:

The setup time (t_{setup}) determines the minimum period of time before the active clock edge in which the signal applied to the D-input has to be stable.

The hold time (t_{hold}) determines the minimum period of time after the active clock edge in which the signal applied to the D-input has to be stable.

If either of these timing requirements is not satisfied the behaviour of the register is considered being undefined. This is indicated by the dashed line in the output waveform on clock edge 3 (which violates the setup time) and 4 (which violates the hold time requirement). On edges 1, 2 and 5 setup- and hold-timing is valid. But because of the timing violations occurring at edge 3 and 4 there is a (non-zero) probability that the register doesn't recover to a stable state fast enough (e.g. in a period of time that is shorter than the clock period). This case is shown at clock edge 5.

Please note that all these timing requirements also hold true for all other synchronous inputs of a register (e.g. set or reset, both not depicted). Chap-

[4] The output of a register is commonly named "Q".

ter 7 of [32] provides detailed information on several possible register implementations and presents timing simulations. More details on registers and their behaviour after timing violations will be given in section 4.2.2.

Nonetheless, the mentioned "black-box view" is valid for synchronous designs because current FPGA toolchains include support for timing driven synthesis (and static timing analysis) and thus take care of all timing requirements of the register.

With this in mind the design mistake can be identified in the following: Both registers in figure 3.5 impose setup- and hold-constraints on the timing of their synchronous inputs related to the free running 50 MHz clock of the EyeBot M6 (which enters the FPGA as `GCLK11`). The analysis will be done for each block individually:

mod_interface The `D` input of the register `FDR` is connected to a constant signal and can therefore be ignored for setup- and hold-timing verification. The (synchronous) reset input `R` though is connected to a signal driven by a combinatorial logic block (`DECODER`). This decoder generates a negated enable signal (nDIO) for the xdio-module by comparing a part of the CPU address (`CPU_A[18:11]`) with a fixed value. The problem comes to existence here because `CPU_A` is generated in the CPU clock domain and therefore will change at arbitrary points in time from the FPGA's point of view. This obviously also is true for the "enable" signal fed into and sampled by the register. If the CPU address (and therefore the "enable" signal nDIO) changes too close to a rising edge on `GCLK11` a setup- or hold-time violation might occur.

xdio Register `FD` and multiplexer `M2_1` together form a register with enable: Only if `S0` is set to `1` a new value (applied to input `D1` of the multiplexer) can reach the register. Otherwise the current value of the register will be stored as "new" value, effectively disabling writing. A similar problem as above can be identified here: The multiplexer is controlled by a signal that is generated from signals originating in the CPU clock domain (`CPU_A[20]`, `CPU_NCS`, `CPU_NWR` and `CPU_A[10:1]`). If one of these signals changes at an unfitting point in time the multiplexer might be in the process of switching between its two inputs when the rising edge of `DCM_CLK150` triggers `FD`. Then, depending on the data and the switching behaviour of the multiplexer a setup- or hold-violation in `FD` might occur. Please note that even a single timing violation in one of the sixteen 1 bit registers may render the complete word useless.

Errors generated by these design mistakes generally are hard to trace because they heavily depend not only on the timing relation of both clocks but also on the delays introduced by combinatorial logic and therefore can not reproduced reliably. Additionally, the implementation of a "register with enable" using a register (without enable) accompanied by a multiplexer is a (little) waste of logic resources because all (hardwired) registers in the FPGA already feature a clock enable-input. Apparently the synthesis tool is not capable of inferring a register with clock-enable when processing VHDL code written on the behavioural level of abstraction used in the old design.

Figure 3.6 shows an exemplary waveform depicting signals both from the CPU clock domain (above dotted blue line) and from the FPGA clock domain (below line). For

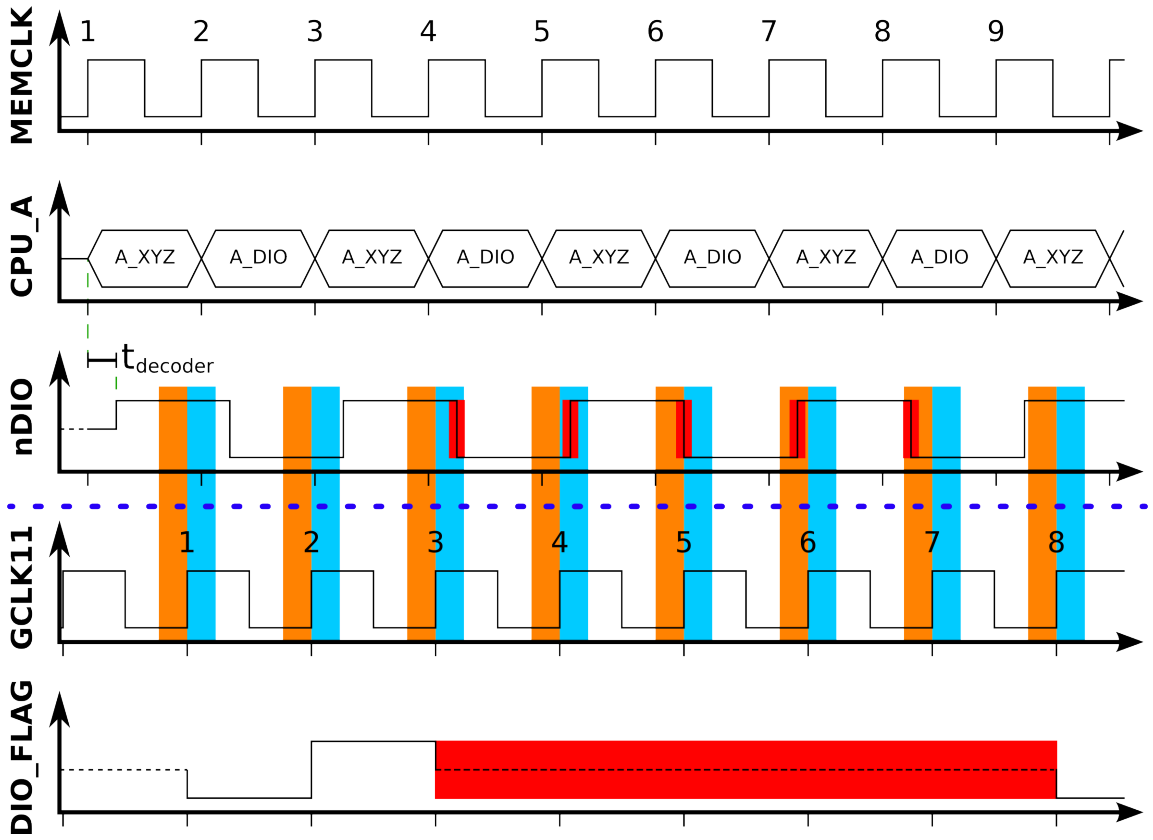

Figure 3.6: Timing violations in the old design

illustration purposes the timing relationships between MEMCLK and GCLK11 and between MEMCLK and CPU_A do not reflect the actual values. (In reality MEMCLK is twice as fast as GCLK11 and CPU_A is stable for more than one MEMCLK – in the figure GCLK11 has been chosen to a frequency close, but not entirely equal, to MEMCLK). These variations do not affect the predications but lead to a more straightforward waveform. Additionally, t_{c2q} of the register is assumed to be negligible.

Timing requirements of the `FDR` register are illustrated in the same way as in figure 3.5, the signal names correspond to the names in figure 3.4. In this example the CPU accesses the xdio module every second `MEMCLK`. Therefore, the address bus (`CPU_A`) is set to the address of the xdio module (`A_DIO`) every second `MEMCLK`. The address on the bus then gets compared to the fixed address `A_DIO`. If the addresses match, `nDIO` will be driven to `0` (after a certain constant delay $t_{decoder}$) which releases the synchronous reset input of `FDR`. On the next rising edge of `GCLK11` the `D` and `R` inputs of the register are sampled and the output `Q` is updated accordingly.

It can be seen that the clock signals have a variable relative skew (the interval between their rising edges is not constant). Because of that the sampling point of the `FDR` register (determined by `GCLK11`) may be close to a signal change on `nDIO` (whose timing is determined by `MEMCLK` and $t_{decoder}$) which then leads to timing-violations ranging from rising

edge 3 to rising edge 7. No timing violation occurs at rising edge 8 and the register is assumed to be able to recover to a defined output value.

Besides that, an additional problem shall be pointed out: Because of these long asynchronous paths (originating at input pins driven by the the CPU) the behaviour of the resulting FPGA design also heavily depends on the placement and routing chosen by the FPGA design tools. If for example the routing delays for each single bit on the bus between `M2_1` and `FD` would vary by a large amount the probability of ending up with at least one wrong bit in `FD` is larger than if all bit lanes would be delayed by the same amount of time.

3.3.3 Identified problems

In summary the following can be put on record:

- The PXA CPU supports both synchronous and asynchronous bus transfer modes[5)]
- Address and data buses are buffered on the PCB, but the transfer qualifier ($\overline{\text{CS}}$, $\overline{\text{OE}}$ and $\overline{\text{PWE}}$) are not
- RD/$\overline{\text{WR}}$ is not connected to the FPGA
- The deployed behavioural coding style somehow hides (in this case important) implementation details and leads to a less efficient implementation
- The old FPGA design ignores the existence of the two clock domains and is therefore prone to timing-violations and their consequences

More details and the solution to this problem can be found in section 4.2.

Apart therefrom another problem endangering the data transfer was identified. It will be presented in the following section.

3.4 Timing constraints

During the exploration of the existing FPGA design another problem was found. An essential part of the FPGA design flow, the applying of input/output timing constrains, has been skipped. These timing constraints are used to inform the FPGA toolchain of the timing properties and requirements of the surrounding logic.

This section first briefly introduces why timing constraints are an essential part of the design flow and then discusses the potential effects of the lack of (I/O) timing constraints.

[5)] meaning that there are modes that not only support connecting clocked and non-clocked devices but also allow an external peripheral to halt the processor if needed

Looking back at section 3.3.2 it can be found that registers impose certain timing requirements to their data inputs to avoid setup- and hold-violations. This not only holds true for registers inside the FPGA but also for registers outside the FPGA that are connected to the latter (e.g. the SRAM). Therefore, the requirements of both the internal and the external registers have to be considered by the toolchain. Obviously, the compliance with these requirements can only be verified if the delays on the numerous paths inside the FPGA are known. Because the routing inside an FPGA is newly created on every run of the toolchain all path delays may change drastically between design iterations.

Three different cases have to be considered:
Assume that a register inside the FPGA is clocked by an external device (e.g. one of the cameras) and samples one output of that external device (which is connected to one of the FPGA's input-pins). If the path delay between data input-pin and register is much longer (or shorter) than the delay on the path connecting the external device's clock to the register's clock-input it might happen that a data transition arrives too close to the active clock edge and therefore violates the register's setup- or hold-requirements. Therefore, the delays on both clock- and data-paths starting at an input-pin of the FPGA have to be monitored closely.
The same challenge arises if an external device is clocked by the FPGA (or an external clock source) and shall receive data synchronized to that clock. If the path carrying the data to the FPGA's output-pin is much longer (or shorter) than the clock network the timing requirements of the external device might get violated. Therefore, the delays on paths originating in and leaving the FPGA have to be monitored as well.
The most obvious paths that have to be checked are the data paths between two registers inside the FPGA. If the delay introduced by the logic between the registers is too large (compared to the clock period), the timing requirements of the second register might get violated or it might even miss the signal transition completely. In the old design the only existing constraint was of that type and covered all register-to-register ("synchronous") paths that were clocked by the 50 MHz oscillator (or a derived clock).

To allow this monitoring the FPGA toolchain has to be informed which path delays should be considered (because it would be unfeasible to monitor and output all potential paths). If the paths have been identified even the requirements can be entered into the toolchain. This not only enables the toolchain to verify whether all requirements have been met but also allows the tools to take these requirements into account when implementing the design. This leads to an optimized placement and routing inside the FPGA.

If no timing constraints are specified the toolchain will place and route the design according to its internal objectives (e.g. "minimum space consumption"). Depending on the design and the tool, its revision and its configuration this might lead to an FPGA design that does not comply with the timing requirements of the surrounding logic.

This problem might have contributed to the problems found in the old design even though one constraint was used. The reason therefore is the large number of asynchronous paths that were not covered by that (synchronous) constraint. It is difficult to prove that the lack of I/O timing constraints is (solely or even only partially) responsible for these problems

because the failing revisions of the old design have not been preserved. The author of this thesis assumes that both the problem identified in the previous section and the missing timing constraints have contributed to the failure of the old design.

In summary the following can be put on record:

- Timing constraints are an essential part of the FPGA design flow
- The lack of timing constraints may lead to the errors observed in the old design

Therefore, I/O timing constraints have been integrated to the design. More details can be found in section 4.3.

3.5 Link-up of cameras, SRAM and FPGA

Looking back at figure 3.1 it can be found that each camera provides a clock signal to the FPGA and therefore has its own clock domain each. The clock signal indicates when the camera data bus is stable and may be latched by the FPGA. The old FPGA design however used an internal clock signal to sample the camera signals and therefore is prone to the same problem already identified in section 3.3.2. If the signals on the camera data bus change close to an edge of the internal clock signal a setup- or hold-violation might occur in the sampling registers. The solution to this problem is based on the findings of section 4.2.4 and can be found in appendix F.1.

In addition, a CDC-related problem was found in the old SRAM interface. The interface was used to buffer an image received from the camera and to make it then available to the CPU. This was implemented in the following way: An internal 100 MHz clock signal of the FPGA was routed to the SRAM's clock input. One bit in a CPU-accessible control register was used to switch between accesses from the internal camera module and accesses from the CPU.
For camera accesses the SRAM's address bus was driven by a counter in the FPGA. This approach is fine from a CDC point of view because both clock and bus signals are related to the internal clock of the FPGA[6)].
CPU read accesses however were implemented by routing the CPU's address bus to the SRAM's address bus while the SRAM still was clocked by the internal 100 MHz clock of the FPGA. This might cause setup- or hold-violations in the address registers of the SRAM that sample the address bus (originating in the CPU-CD) using the clock signal from the FPGA-CD. The reason therefore is that these two CDs are not related to each other (and thus will have an arbitrary phase relation).

[6)] Even though both clock and address bus are generated in the same CD additional care has to be taken because of the FPGA's internal routing delays. More information can be found in section 4.4.2.

4 Development of an image processing framework for the FPGA

4.1 Overview

This chapter describes all the actions that have been taken to solve the problems identified in chapter 3. It is organized in three sections:

Bus bridge (section 4.2): Based on the findings of section 3.3 a bus bridge was developed that shall overcome all the limitations and problems of the old FPGA design's interface. The section contains not only additional low-level information about register but also how synchronization between two clock domains (CDs) can be achieved in general. Based thereon, the bus bridge, its features and all design decisions made will be presented.

Timing constraints (section 4.3): As described in section 3.4 the old existing FPGA design was lacking timing constraints for both CPU and camera interface. This section describes not only the timing constraints needed for these two interfaces but also derives some additional ones used to enforce a proper routing of the bus bridge. The latter ones were needed to limit delays on certain asynchronous paths that were not covered by regular (synchronous) constraints.

SRAM controller (section 4.4): The SRAM interface of the old design (already mentioned in section 3.5) was only intended for temporary buffering between camera module and CPU. The interface only supported alternating accesses from either camera module or CPU and was controlled by software. The stereo image processing system developed by a fellow student (see [29] for more information) however requires multiple quasi-simultaneous accesses to the SRAM. Therefore, a new SRAM controller was designed to meet this requirement.
Sadly a time-consuming problem on PCB-level (related to the clocking of the SRAM) was found during research and rendered the implementation of the SRAM controller impossible. The section anyhow explains the proposed implementation, the ideas behind and demonstrates its interaction with the image processing system. In addition, the problem is analyzed and a solution therefor is presented.

During the development of the bus bridge two major goals were pursued: Firstly, reliable (and fast) communication between FPGA and CPU should be established. Secondly, as much as possible of the complexity of the asynchronous interfaces should be encapsulated

to unburden other developers from dealing with this low-level kind of interfacing. This was achieved by replacing all partially asynchronous logic inside the FPGA with synchronous circuits which are not only easier to understand but also fully supported by the FPGA design toolchain. Additionally, this synchronous approach improves the value of timing simulations (because a simulation of a synchronous circuit generally is more likely to reflect reality compared to a simulation of an asynchronous circuit).

4.2 Interfacing the CPU: The bus bridge

4.2.1 Conceivable approaches to interface CPU and FPGA

This section first theoretically introduces two straightforward (but still sophisticated) approaches to interface the FPGA to the CPU. The goal of these approaches is to avoid timing-violations both in the CPU and the FPGA. This can be achieved by ensuring certain (known) timing-relationships between one or more clocks and all status- and data-lines. Sadly none of these concepts could be used on the EyeBot M6 so an alternative, matched (but also more complex) solution had to be designed.

4.2.1.1 One clock domain: FPGA synchronous to CPU

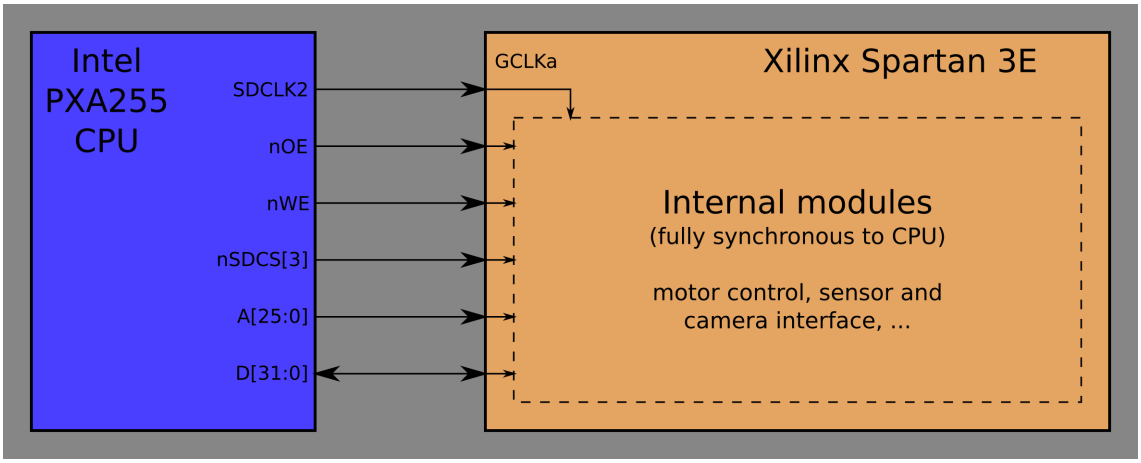

Figure 4.1: Fully synchronous system (FPGA using clock from CPU)

Figure 4.1 shows the most obvious approach: One of the FPGA's global clock inputs (GCLK) is connected to one clock output of the CPU's memory controller (e.g. SDCLK[2]). The memory controller then has to be configured in a way that this clock output emits a continuous non-gated clock[1)]. This clock (generated by the CPU) will

[1)] For anyone interested: The relevant bits are K2FREE, K2RUN and APD in the MDREFR register of the controller. More information can be found in table 6-5 of [5].

then be fed into a DCM of the FPGA and all clocks needed inside the FPGA will be derived from it. This approach makes use of the "Synchronous static memory" interface of the CPU, therefore the memory controller has to programmed to "Non-SDRAM mode" to avoid the generation of SDRAM-commands (like Bank Activate, Precharge, etc.).

This approach has multiple advantages:

- There is no need for an external clock source (e.g. an oscillator). Additional (slower or faster) clocks can easily be generated by a DCM inside the FPGA.
- All bus signals driven by the CPU will be synchronous to the clock leaving the CPU. Inside the FPGA this clock then can be used to sample all these bus signals. Because both registers inside the CPU (driving the bus signals) and registers inside the FPGA (sampling the bus signals) are clocked by the same clock it can be ensured that no timing-violations will occur in any of these registers[2)].
- The whole system therefore forms one single CD. There is no need for any kind of resynchronization or clock domain crossing which keeps the system easy to understand and simplifies maintenance.
- Depending on the chosen configuration the SDCLK frequency can be as high as 50 MHz. A (16 bit) data word can be transferred on every rising edge. Assuming a burst transfer of infinite length a notional maximum data transfer rate of 100 MB/s can be achieved.

4.2.1.2 Two clock domains, CPU clock available to FPGA

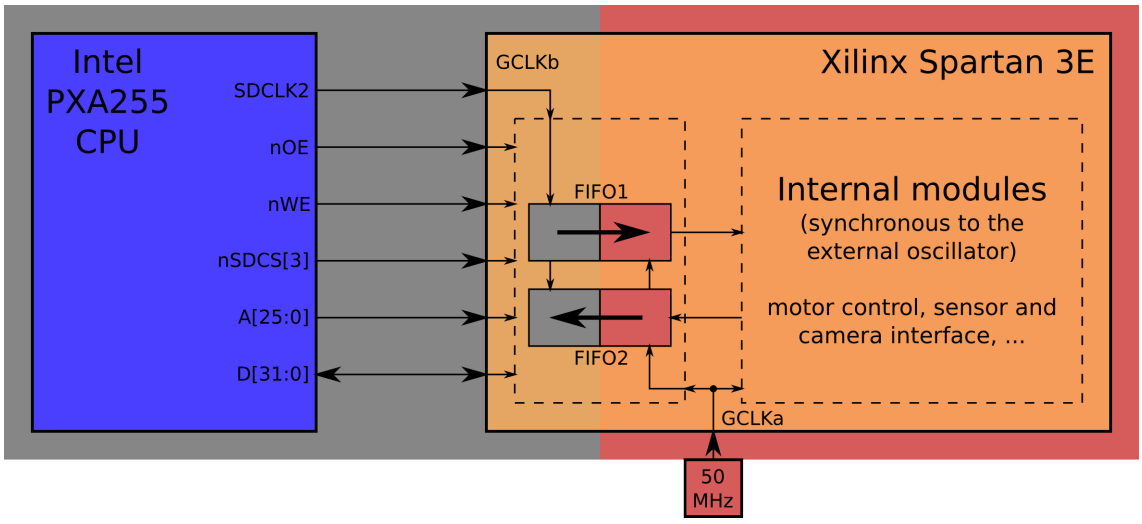

Figure 4.2: Synchronous system with two clocks (CPU clock available to FPGA)

[2)] The input/output-circuitry at every IO-pad of the FPGA and the DCMs contain special features (delay-lines and deskew-units) to balance timing in a way that no timing-violations will occur.

Figure 4.2 shows another valid approach that can be used if the FPGA shall be clocked by an external oscillator[3]. This oscillator and again one clock output of the CPU's memory controller are connected to two of the FPGA's global clock inputs. This leads to a system with two clock domains that meet inside the FPGA. To transfer the signals coming from or going to the CPU clock domain a special type of FIFO is used inside the FPGA, the so called "independent clock FIFO". It not only serves as a regular queue (as every FIFO) but also is able to pass over data between two (completely independent) clock domains. Because this type of FIFO is one of the only two circuits that allow the safe crossing of clock domains it will be examined in-depth in section 4.2.4.

Compared to the first approach (with only one CD) a few differences can be observed:

- Two dedicated FIFOs are needed to transfer the data between CPU and FPGA clock domain in a safe fashion. These FIFOs take up additional CLBs and at least one Block-RAM each. Additionally, some logic has to be inserted on the FPGA side of the receiving FIFO (FIFO1) to recreate some kind of bus transactions for the inner modules.

- The FIFOs introduce (variable) delay on the path between CPU and FPGA. The actual value heavily depends on the design of the FIFO, the frequencies of the two clocks and their phase relationship. This delay mostly is created by the circuitry used to synchronize signals between the two clock domains.

- Write accesses can be handled easily: After detecting an access to the FPGA both address and the data to be written will be stored in FIFO1. Both then will be driven to the internal bus and the accessed module will react appropriately (e.g. by updating one of its internal registers).

- The delay introduced by the FIFOs has huge impact on CPU reads that access resources in the FPGA: During a regular VLIO read access (see figure 3.3 in section 3.3.1) the CPU expects the read data to be on the data bus at a certain time after driving the address bus and asserting $\overline{\text{CS}}$ and $\overline{\text{OE}}$. When using this FIFO-based approach all the following has to happen during that period of time: Firstly, a control word ("request") indicating a read access and containing the requested address has to be written to FIFO1. Then FIFO1 has to transfer this request to the FPGA clock domain. After that, an access to the internal bus has to be triggered which then will be answered by the internal module. This response finally has to be sent back to the CPU by storing it into FIFO2 and driving the data bus after the answer has reached the CPU clock domain. The delays introduced by the two FIFOs are in the range of a few clocks (of the corresponding domain) each. Taking this into consideration it becomes clear that it is impossible to drive the data bus fast enough when using the timing shown in figure 3.3, because there are only three clock cycles between the assertion of $\overline{\text{OE}}$ and the point in time when the CPU sam-

[3] This approach can be used if the CPU should be able to enter some sort of power save mode (in which the clock frequency may change or the clock even may stop completely) and the FPGA shall be active all the time.

ples the data bus[4]. Therefore, the bus timing has to be slowed down accordingly (by changing the configuration of the memory controller) which will affect (both read and write) performance.

4.2.1.3 On the EyeBot M6: Two clock domains, CPU clock not available to FPGA

Unfortunately these two approaches above are not applicable to the EyeBot M6 because there is no clock signal leaving the gumstix PCB that could be connected to the FPGA (see figure 3.1 in section 3.3.1 for reference). Figure 4.3 shows the situation on the EyeBot M6.

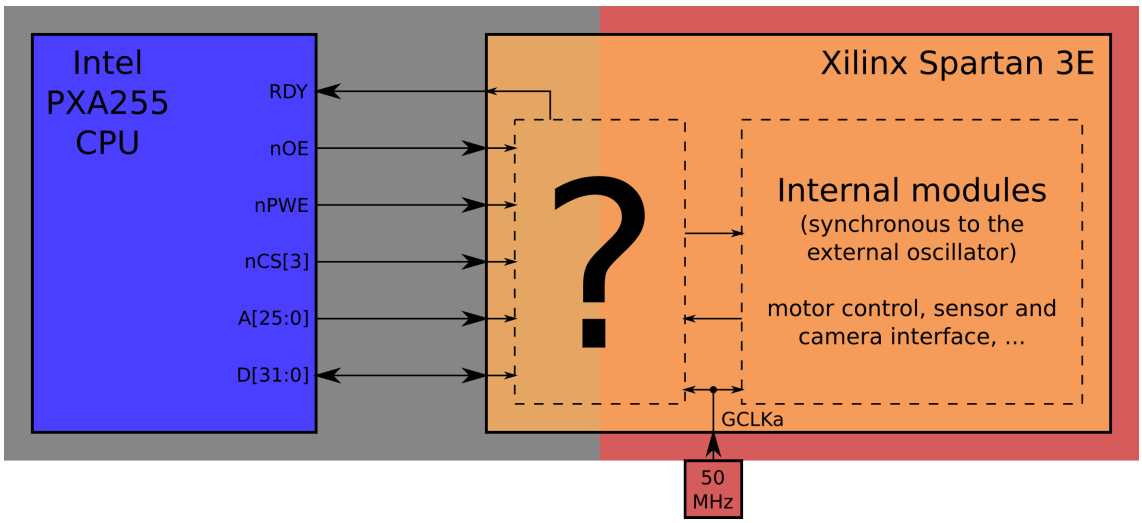

Figure 4.3: Asynchronous system (CPU clock unknown)

In summary it can be said that

- plain sampling can not be used because the FPGA has its own independent clock and therefore the signals driven by the CPU will change asynchronously and that
- crossing the clock domains using an independent clock FIFO can not be used because no CPU clock is available to drive one side of the FIFO.

Therefore, a new approach to handle this asynchronous interface has to be found.

4.2.2 Metastability

The previous section (4.2.1) demonstrated that there is no straightforward possibility to interface CPU and FPGA because of the lack of a clock signal from the CPU. Because FPGAs are fully synchronous ICs the asynchronous CPU bus somehow has to be

[4] By prefetching the requested data at the assertion of $\overline{CS}$ (before it is actually known whether a write- or a read-access will be initiated) two additional clock cycles could be gained. But even five clock cycles would be too short.

synchronized to the FPGA-CD. Since registers are the only elements that are capable thereto the impact of timing-violations has to be probed further. The following therefore focuses on the possible consequence, the so called metastability. Based thereon, a classical synchronization circuit will be introduced in section 4.2.3.

This section starts with an introduction to metastability and recapitulates why it can not be avoided when interfacing asynchronous signals. After that, the possible outcomes of a timing-violation will be presented. The section concludes with different theoretical methods to characterize metastability.

4.2.2.1 Introduction

As already noted in section 3.3.2 (figure 3.5) registers require their inputs to adhere to certain timing constraints. If these requirements are violated the output of the register will be unpredictable. In synchronous circuits this behaviour is avoided by ensuring that all timing-requirements for each register are met by design. To avoid setup-violations the maximum combinatorial path delay $t_{logic,max}$ (between two registers) has to be smaller than the clock period T_{clk} (or – in case of multi-cycle designs – a multiple thereof) minus t_{c2q} and t_{setup} of the registers ($t_{logic,max} \leq T_{clk} - t_{c2q} - t_{setup}$). To avoid hold-violations the minimum combinatorial path delay $t_{logic,min}$ has to be longer then t_{hold} minus t_{c2q} of the registers ($t_{logic,min} \geq t_{hold} - t_{c2q}$). Besides that, the skew of the whole clock distribution network has to be taken into consideration. Unfortunately these checks are infeasible when interfacing an asynchronous signal because no timing relation is known between sending and receiving register.

When a signal is sampled in another clock domain than it was created in it is impossible to avoid these timing violations because the signal may change at an arbitrary point in time. Assume signal S_a is generated in CD_a and has to be transferred to CD_b. Transitions on signal S_a will always happen after a certain (maximum) delay after the active edge of CLK_a. If CLK_b is not derived from or by some other means synchronized to CLK_a the phase offset between the two clocks will change over time. This will induce setup- or hold-violations on (at least) the first register in CD_b (see figure 3.6 for reference) depending on the exact timing of both the circuit and the two clocks. Because these violations neither can be avoided nor predicted in time a way to handle them appropriately has to be found. This has to be done on register- or even transistor-level what generally leads to specialized circuits and even to handcrafted transistor stages in ASIC designs.

4.2.2.2 Possible failure modes

To analyze the requirements for a synchronization circuit first all possible failure modes of a single register have to be identified. If the timing requirements of a register are violated its output is considered to be undefined. Whilst this (high) level of abstraction is perfectly valid for behavioural simulation it does not reflect the physical reality because a voltage level of “undefined V” does not exist. In reality the output of the register might

- stay unchanged,
- immediately adopt the logic value of the input pin before[5] or after its transition or
- enter a metastable state

after the timing requirements of the register have been violated.

Please note that the term "metastability" is not well-defined: Some authors consider a register being "metastable" as soon as its (setup- and hold-) timing requirements are violated, while others use the term only to describe situations when the clock-to-output delay t_{c2q} of the register is longer than specified in the data sheet. Finally, the term "metastable" is also used to denote the operating point of a feedback loop formed of two inverters (e.g. a latch) in which both their outputs are at the same electrical level. Theoretically a latch would stay in this state for an infinite amount of time ($t_{c2q} \rightarrow \infty$), but in real circuits a small disturbance (e.g. noise or voltage variation) will push the circuit out of equilibrium.

In this thesis the second definition will be used: Metastability has occurred if t_{c2q} is longer than specified in the data sheet.

The first two possibilities do not pose a problem because the output will be a stable legal logic value of either `0` or `1`. It is not important which value appears at the output because the input signal was changing anyhow so both `0` or `1` are considered correct.

The third outcome tough imposes a serious problem for digital circuits because the exact behaviour of a metastable circuit can not be predicted. The output of a metastable register might

1. finally settle to `0` or
2. settle to `1` (after a certain amount of time),
3. oscillate in a voltage range exceeding the trigger limits of the logic family and finally settle to a stable level or
4. oscillate in a voltage range smaller than the trigger limits of the logic family and settle to a stable level (after a certain amount of time),
5. finally change to the value of the input or
6. change to the negated value of the input (after a certain amount of time).

The problem in all this cases is that the output of the register will change its logic level spontaneously (for one or even several times). This change will happen a certain time after the active clock edge and this interval will be longer than the clock-to-output delay specified in the data sheet. This additional delay can be considered as "time that is needed by the register to decide whether the (changing) input is considered being a logic `1` or a logic `0`". Case 3 additionally imposes a limit on the usage of register outputs that are endangered to metastability: An output of such a register must never be used to drive

[5] The input might have changed after it was sampled one clock period before.

any clock inputs (e.g. of other registers) because an oscillation on that net might lead to switching activity at unwanted points in time.

4.2.2.3 Characterizing metastability

The actual behaviour of the register can only be described by statistical means because the underlying physical processes are depending on values that can be characterized statistically only, too. Two important values can be defined:

The metastability window determines the length of the interval around the active clock edge in which a transition on the data input forces a register into a (theoretically) persistent metastable state. The actual length and the position of that window depend on used IC technology parameters, temperature, voltage and noise levels.

The resolution time is the time that is taken by a register before it finally settles to a stable output value after the metastability window has been hit. Theoretically this interval could be of infinite length if the metastable operating point was reached perfectly. Practically the resolution time mainly depends on the supply voltage and on the voltage levels on the feedback paths just after the active clock edge. The latter ones obviously are influenced by the exact shape of both clock- and data-pulses and again by IC technology parameters, temperature, voltage and noise levels.

This topic is covered by various publications: [16] presents real life measurements of the metastable recovery time in Virtex 2-FPGAs and extrapolates MTBF values based thereon. The article states that the metastable behaviour of a real register (in an FPGA) can be quantified by likelihood measurements only because it is impossible to build a circuit that is accurate enough to hit the metastability window in a reliable way. Therefore, two asynchronous clocks were used to push a register Q_a into metastability. The number of metastable events in the register Q_a was determined by sampling its output S_a at two different points in time after the active (rising) clock edge and by comparing both samples. The sampling was implemented by two registers, one clocked on the falling (Q_c) and one clocked on the rising edge of the clock (Q_b). If the clock-to-output delay t_{c2q} of register Q_a has increased by more than half a clock period the transition on S_a will happen after the falling clock edge and will therefore be missed by Q_c. Assuming that t_{c2q} is smaller than the clock period (which is very likely) Q_b will sample S_a after its transition. Therefore, S_c and S_b (which are the outputs of Q_c and Q_b resp.) will have different logic values. This state then can be used to trigger a counter that counts the number of metastability events in Q_a. The interval in which Q_a will be checked for metastability can be adjusted by setting the clock period to different values. By increasing this period additional time to resolve the metastable state can be granted to Q_a. A period of time provided to a register to leave a metastable state is frequently called (allowed) settling time (see [11] or [18]).
The most important result of the above measurements is that even a small increase in the allowed settling time has a huge positive effect on the MTBF of the register ([13]). From this it follows that the probability of an output transition closer to the clock edge

is higher than the probability of an output transition further away from the clock edge. In other words: The probability for a short resolution time is much higher than the probability for a longer resolution time. Additionally, the measurements are indicative of an improvement of the metastability behaviour (in terms of a higher MTBF) when comparing a Virtex 2-FPGA with an older XC4005. It has to be pointed out though that according to [18] the metastability behaviour of newer Virtex-FPGAs got worse for unknown reason. The above and the complete lack of (official and unofficial) numbers for the Spartan architecture lead the author of this thesis to stop further investigations into metastability estimations on the Spartan-3E.

[31] however presents mathematical models to estimate the behaviour of metastable CMOS circuits. It also discusses methods to reduce the probability of errors induced by metastability both on transistor- and system-level. Equation 4.1 shows (a slightly adapted version of) equation 3.42 on page 55 of [31]. It is the result of an extensive derivation and allows the quantitative computation of the MTBF. A "failure" here is defined as "the metastable state lasts longer than a certain, chosen time" which is identical to "the register fails to resolve the metastable state within a certain, chosen time". This time will be called the metastable time t_m and is measured from the active clock edge. Because metastability lengthens (the observable) t_{c2q} it is obvious that t_m is always greater than t_{c2q} specified in the data sheet.

$$\text{MTBF} = \frac{e^{\frac{t_m - \overline{t_{pDL}}}{\tau}}}{f_{clk} f_{data} t_w} = \frac{e^{\frac{t_s}{\tau}}}{f_{clk} f_{data} t_w} \text{ with } t_s = t_m - t_{pDL} \tag{4.1}$$

Equation 4.1 also is a more detailed version of the equation in [16], page 2. In the following the symbols used in both will be introduced and the qualitative relationship between MTBF and additional settling time will be shown.

τ is the latch time constant and t_w is the metastability window already introduced earlier[6]. Both of them are technology parameters depending on load capacitance, transistor gain, threshold and supply voltage. f_{clk} is the clock frequency that clocks the register under test while f_{data} is the test frequency applied to the data input.
$\overline{t_{pDL}}$ is the average propagation delay of the underlying latch structures. The difference between t_m and $\overline{t_{pDL}}$ can be interpreted as additional settling time t_s that is granted to the register to avoid that a metastable state will be entered all too often (which would be equivalent to a too small MTBF).

The measurements in [16] were performed by observing the MTBFs at two different frequencies f_{clk}. Because both frequencies and therefore both (half-)periods were known the difference of the two allowed settling times (Δt_s) could be calculated. Based thereon, τ

[6] In [16] the metastability window is named K1 and the inverse of the latch time constant τ is called K2. The comment that "some researchers list 1/K2 as a time constant, (τ)" is somehow misleading because the letter τ in that document actually stands for the allowed settling time.

could be determined using equation 4.2[7)].

$$\tau = \frac{\Delta t_s}{\ln \frac{MTBF_1 f_1}{MTBF_2 f_2}} \tag{4.2}$$

To evaluate the impact of additional settling time on the MTBF the same system will be considered twice ($f_1 = f_2$). Rearranging equation 4.2 leads to equation 4.3 which shows that a small variation on Δt_s will have an exponential effect on the MTBF.

$$\frac{MTBF_1}{MTBF_2} = e^{\frac{\Delta t_s}{\tau}} \tag{4.3}$$

This leads to the following rule of thumb: To reduce the probability of metastability-related errors (identical to a long MTBF) as much as possible additional settling time has to be granted to all registers whose inputs are driven by asynchronous signals. Because the technology parameter τ is unknown for the Spartan architecture no quantitative values can be presented.

Insert E of [32] focuses on timing-requirements from an RTL point of view. Both [31] and [32] include an alternative, incisive representation of the (simulated) effects of metastability on the output of a register. Figure 4.4 reprints figure E-4 from page 433

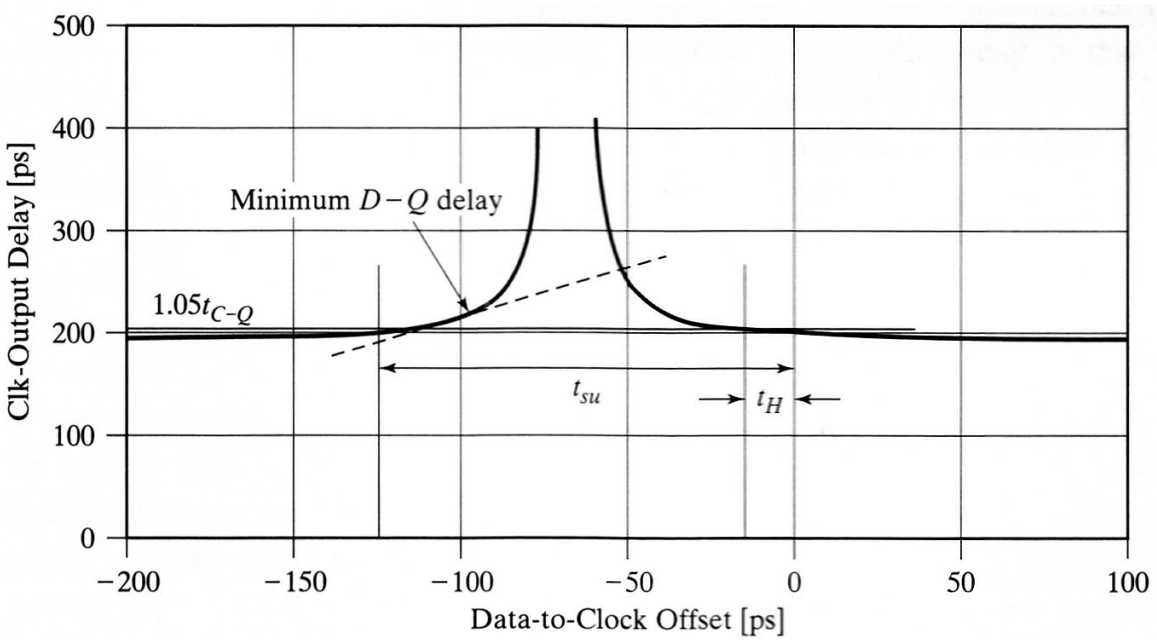

Figure 4.4: Clock-to-output delay as function of the data-to-clock offset (source: [32], page 433)

[7)] Please note that his equation was reconstructed by the author of this thesis because [16] just presents numbers without calculations. Apparently the values in table 1 of [16] were achieved assuming that $f_1/f_2 \approx 1$.

of [32]. It depicts the clock-to-output delay t_{c2q} (see figure 3.5 for reference) as function of the offset between the last change on the input signal and the active clock edge (the data-to-clock offset t_{d2c}). Again an exponential characteristic can be identified. It can be seen that t_{c2q} differs from the "usual" value (below 200 ps) for $t_{d2c} \in [-125; -15]$ ps. Whenever the data input signal changes during that window the signal transition on the output of the register will be delayed. Because of the internal delays of the circuit the window is not centered around 0 ps, which explains why setup times are usually longer than hold times. Setup- and hold-times (labeled t_{su} resp. t_H) were defined by choosing a "tolerable delay" of 5% (tagged by the horizontal line labeled $1.05t_{C-Q}$). It can be seen that (in this example) the hold time t_H is negative because the clock-to-output delay decreased to $1.05t_{C-Q}$ left of $t_{d2c} = 0$ ps. This means that there is no need to hold the data-input on a stable level after the clock edge has occurred.

Please note that (at least in Xilinx FPGAs) in most cases a setup- or hold-time-violation does not induce metastable behaviour because the metastability window is much shorter than the sum of setup- and hold-times recorded in the data sheets (see [15] or [11] for reference). The reason therefore is that IC manufacturers tend to specify setup- and hold-time requirements longer than actually required. This "safety margin" then may be used to compensate process, voltage and temperature variations, routing delays and more.

4.2.3 Clock domain crossing with the two-stage synchronizer

Aside from the already mentioned independent clock FIFO another class of circuits exists that transfers signals between different clock domains. These circuits are called synchronizers and facilitate a (reasonably safe) way to transfer single-bit signals from one to another clock domain.
This chapter firstly presents a classical synchronization circuit, the two-stage synchronizer, its limitations and its applications. The following section (4.2.4) then focuses to the independent clock FIFO which makes use of the two-stage synchronizer.

4.2.3.1 Fundamentals

As shown in section 4.2.2.3 metastability can not be avoided if a register $\mathrm{Q_a}$ receives a signal originating in a different CD. The basic approach pursued is to reduce both probability and impact of a metastability event on the whole system. The circuit used to reach this goal is called a synchronizer.

Equation 4.3 indicates that the mean time between two metastable events leaving a register $\mathrm{Q_a}$ can be increased by granting additional settling time to the register. The latter is the equivalent of maximizing the available slack on the path driven by $\mathrm{Q_a}$. The slack on a synchronous path is defined as $t_{slack} = T_{clk} - t_{c2q} - t_{interconnect} - t_{logic} - t_{setup}$. The slack can therefore be maximized simply by connecting nothing but a second register $\mathrm{Q_b}$ to the output of $\mathrm{Q_a}$ (which is equivalent to $t_{logic} = 0$). The delay caused by the interconnect between the two registers ($t_{interconnect}$) has to be minimized by placing $\mathrm{Q_a}$ and $\mathrm{Q_b}$ close

to each other. Because all available slack should be used as settling time for Q_a t_s can be specified as $t_s = T_{clk} - t_{c2q} - t_{interconnect} - t_{setup}$.

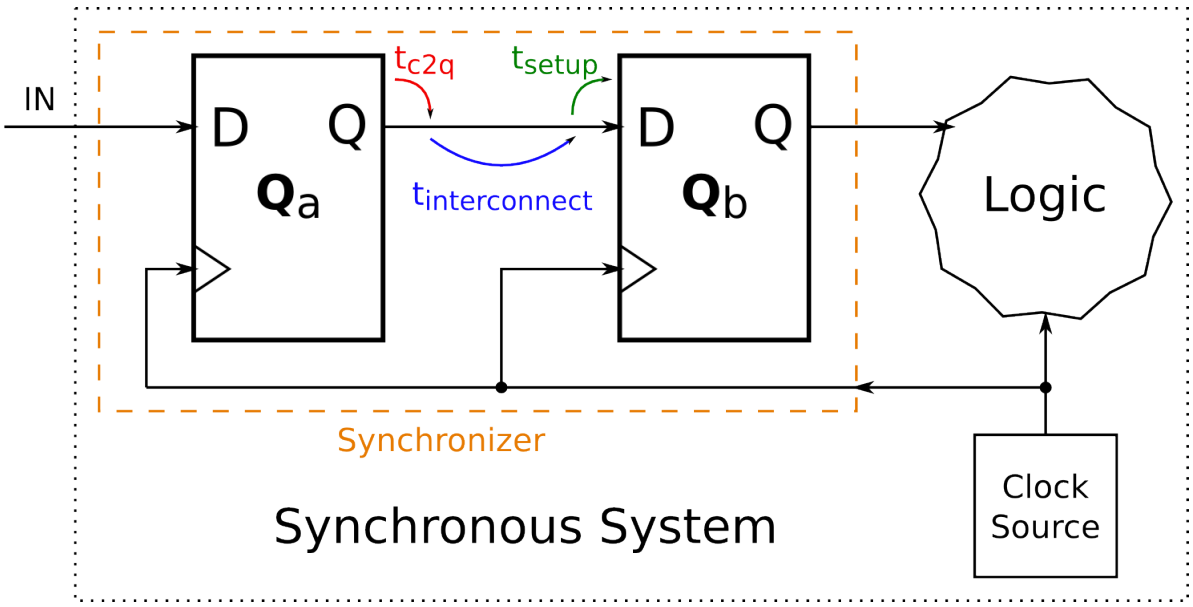

Figure 4.5: The two-stage synchronizer

This simple circuit (shown in figure 4.5) is called two-stage synchronizer. If implemented carefully it is able to extend the mean time between two metastable events by magnitudes. The absolute value of the synchronizer's MTBF still depends on the technology parameters and on the frequencies of the clock (f_{clk}) and the data (f_{data}). Please note that the MTBF of Q_a is no longer reciprocally proportional to f_{clk} because t_s now is a function of $f_{clk} = 1/T_{clk}$. This is in contrast to equation 4.1 (section 4.2.2.3) where the MTBF is reciprocally proportional to f_{clk} (because t_s is assumed to be constant).

Please also note that the probability for a metastable signal at the synchronizer's output is still non-zero. This is the case because Q_a might resolve its metastable state in such a way that the transition will hit the (very small) metastability window of Q_b. The only chance to completely avoid metastable failures (which is identical to MTBF $\rightarrow \infty$) is to grant an infinite settling time to the synchronizer which is obviously impractical for real world circuits.
If the MTBF achieved by the two-stage synchronizer is only a few magnitudes lower than desired value an additional register Q_c can be connected to Q_b. By again maximizing the slack on the path between Q_b and Q_c the cumulative settling time can be doubled. This leads to an increased MTBF at the cost of one additional clock cycle of latency.

Because synchronizing circuits work on the principle of granting additional settling time to one or more registers it is unavoidable that the asynchronous signal handled by the synchronizer will be delayed by (at least) a certain time. Because a metastable event would delay the result by an unknown (theoretically infinite) interval it will be assumed in the following that the asynchronous input signal will not violate setup- and hold-time

requirements of the first register (and therefore metastability will not occur).
In case of the two-stage synchronizer the best- and the worst-case delay can be determined as follows:

Best case: The asynchronous signal changes as close as t_{setup} allows ahead of the next active clock edge. Q_a will then sample the new value on this clock edge. Q_b will see the updated value one clock cycle later and output it delayed by t_{c2q}. The best-case delay therefore can be specified as $t_{sync,min} = t_{setup} + T_{clk} + t_{c2q}$.

Worst case: The asynchronous signal changes as close as t_{hold} allows after an active clock edge. Q_a will sample this transition on the next active clock edge which happens after $T_{clk} - t_{hold}$. Q_b will see the updated value one clock cycle later and will output it delayed by t_{c2q}. The worse-case delay therefore can be specified as $t_{sync,max} = T_{clk} - t_{hold} + T_{clk} + t_{c2q} = 2T_{clk} - t_{hold} + t_{c2q}$.

This uncertainty of the delay introduced by the synchronizer leads to a few problems that have to be handled on system-level. Please note that these problem are not related to metastable events at all, neither inside the synchronizer nor in the downstream logic.

4.2.3.2 Pitfall 1: Synchronization of a single-bit signal

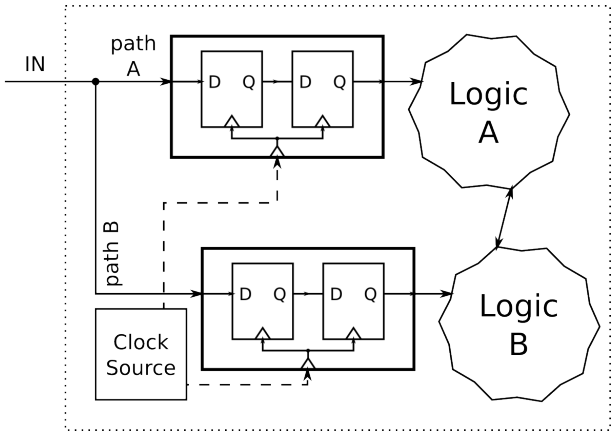

Figure 4.6: Two logic units with one local synchronizer per unit

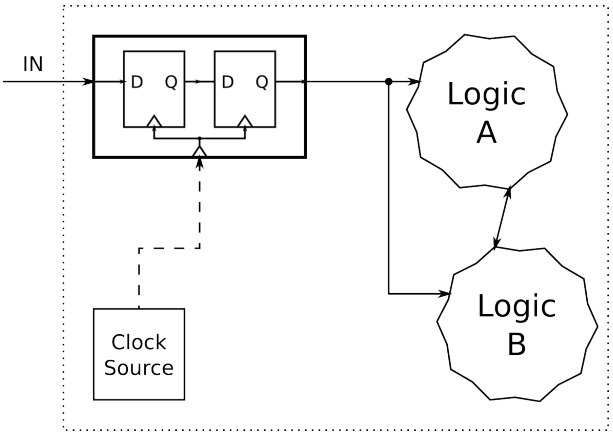

Figure 4.7: Two logic units with centralized synchronization at the input

Consider the following scenario: Two logic units in a synchronous environment depend on one and the same (external) asynchronous input signal `IN`. To increase the MTBF the designer chose to synchronize that signal using two independent synchronizers just before it is fed into the logic units (figure 4.6). This structure is fine from a metastability point of view because the synchronizers (again only if implemented carefully) will push the MTBF by magnitudes. We will further assume that the achieved MTBF has been scrutinized and considered sufficient by the designer. But because of the asynchronous nature of `IN` and because of the uncertainties of the delay introduced by the synchronizers

the system is vulnerable to a race condition: The paths carrying the external signal from the device's input pin to the two synchronizers will never be of the exact same length which results in different propagation delays (path A vs. path B). Therefore, a transition on `IN` will reach the two synchronizers at different points in time. If a clock event takes place just (t_{setup}) after the signal transition on `IN` has reached the upper synchronizer the lower one will miss the transition because the new value is still on the way on path B. In this case an "impossible state" will be observed by the logic because the outputs of both synchronizers are expected to be equal all times.
From a high-level point of view the skew on `IN` would have to be larger than $t_{setup} + t_{hold}$ (so that the upper synchronizer's setup- and the lower synchronizer's hold time requirements will not get violated). In reality both t_{setup} and t_{hold} are specified much longer than needed so the skew on `IN` needed to trigger this scenario might be in the magnitude of picoseconds or even lower.
The solution (depicted in figure 4.7) is straightforward: Only one central synchronizer transfers `IN` into the system's CD. Assuming that the synchronizer did not enter a metastable state its output is synchronous to the clock. Therefore, the design tools are able to verify that a transition on the synchronized `IN` safely reaches both logic A and B before the next active clock edge. This leads to

Rule 1: An asynchronous input signal must not be synchronized by more than one synchronizer. This avoid race conditions caused by divergent synchronization results.

The same rule is valid for designs that use a single register instead of the two-stage synchronizer (figure 4.8) which might be an appropriate approach for slow systems interfacing an even slower asynchronous input.

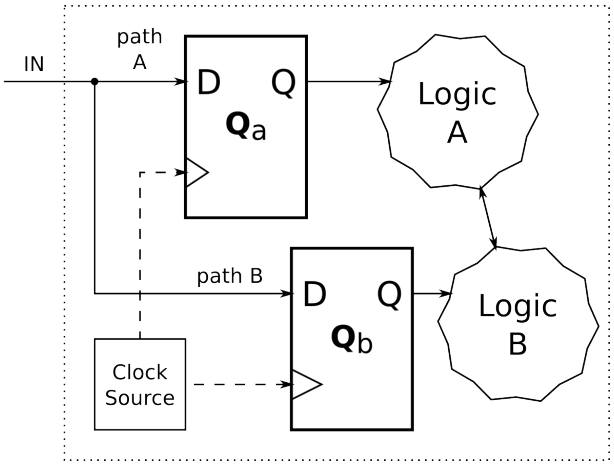

Figure 4.8: Two registers sampling an asynchronous input signal

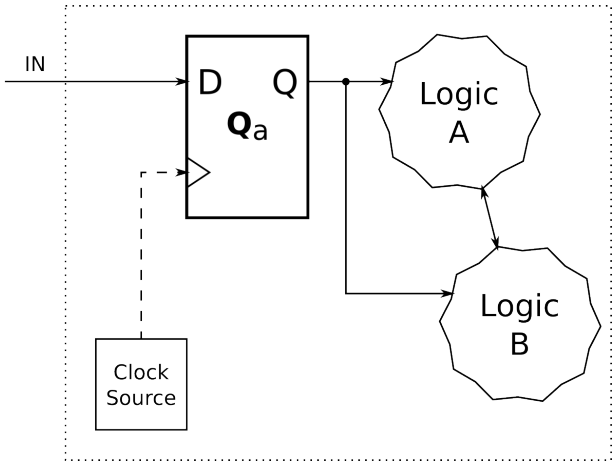

Figure 4.9: Asynchronous input signal sampled by one register

Looking back at figure 3.4 (section 3.3.2) exactly this scenario can be identified in the `xdio`-module: `CPU_NCS`, `CPU_NWR` and `CPU_A` are used to generate a write-enable signal for each submodule that is used to control the multiplexer `M2_1` in front of the 16 bit register `FD`. Hence there are paths from an asynchronous input (e.g. `CPU_NWR`) to more that one register

that indirectly synchronize that input to the FPGA's CD. If the timing relationship between a transition on `CPU_NWR` and the active clock edge (on `DCM_CLK150`) is unlucky the multiplexer might still be in the process of switching between its inputs when `FD` samples its inputs. Therefore, only some bits in `FD` might be updated to the new value whilst others would stay unchanged.
The solution is the same as already presented above: The asynchronous input signal (here: `CPU_NWR`) has to be synchronized at a central location before it is fed into subsequent logic stages (figure 4.9).
Please note that this is exactly the same type of race condition as described above, the only difference is the higher MTBF of the circuit presented in figure 4.7.

Another pitfall inherent to flag transfers across CDs has to be considered: Because of the unknown relationship between the clock signals of sending and receiving (and therefore synchronizing) CDs special care has to be taken to ensure the safe reception of the flag in the destination CD.

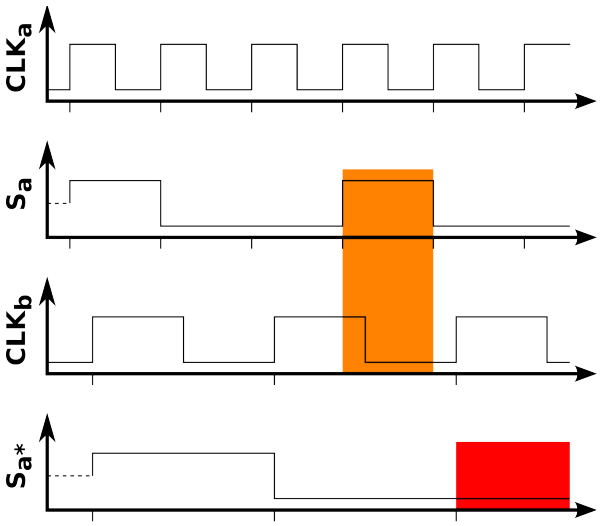

Figure 4.10: Flag synchronization failed

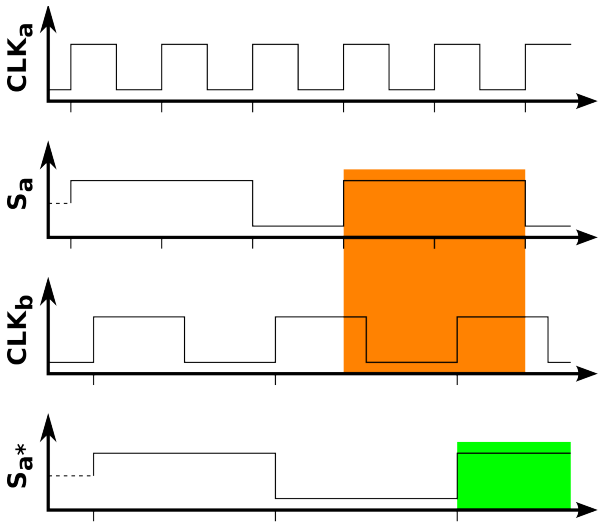

Figure 4.11: Flag synchronization successful

Figure 4.10 shows a typical example of two clock domains running at different frequencies. Signal S_a is driven by CD_a to flag a special state and shall be transferred to CD_b. S_a* is the waveform observed after the first register inside the synchronizer in CD_b. It can be seen that the flag-pulse might not be recognized by the destination CD depending on (frequency- and) phase-relation of the two clocks.
The solution depicted in figure 4.11 is based on the knowledge that the frequency of CLK_a is twice the frequency of CLK_b: If S_a is asserted for at least two[8)] CD_a clock cycles the flag-pulse will be detected by CD_b. This scenario obviously only shows up when transferring a signal from a faster to a slower CD.

If the clock relation between the two CDs is unknown a more complex solution has to be de-

[8)] If the clock jitter of one or both clocks is not negligible S_a has to be asserted for at least three CD_a clock periods which is equal to 1.5 CD_b clock periods.

ployed: By implementing a request/acknowledge-protocol as depicted in figure 4.12 CD_a will assert REQ until CD_b acknowledges the reception of the flag by asserting a signal ACK which will then be recognized by CD_a.
Because REQ resp. ACK will be generated in CD_a resp. CD_b and read in CD_b resp.

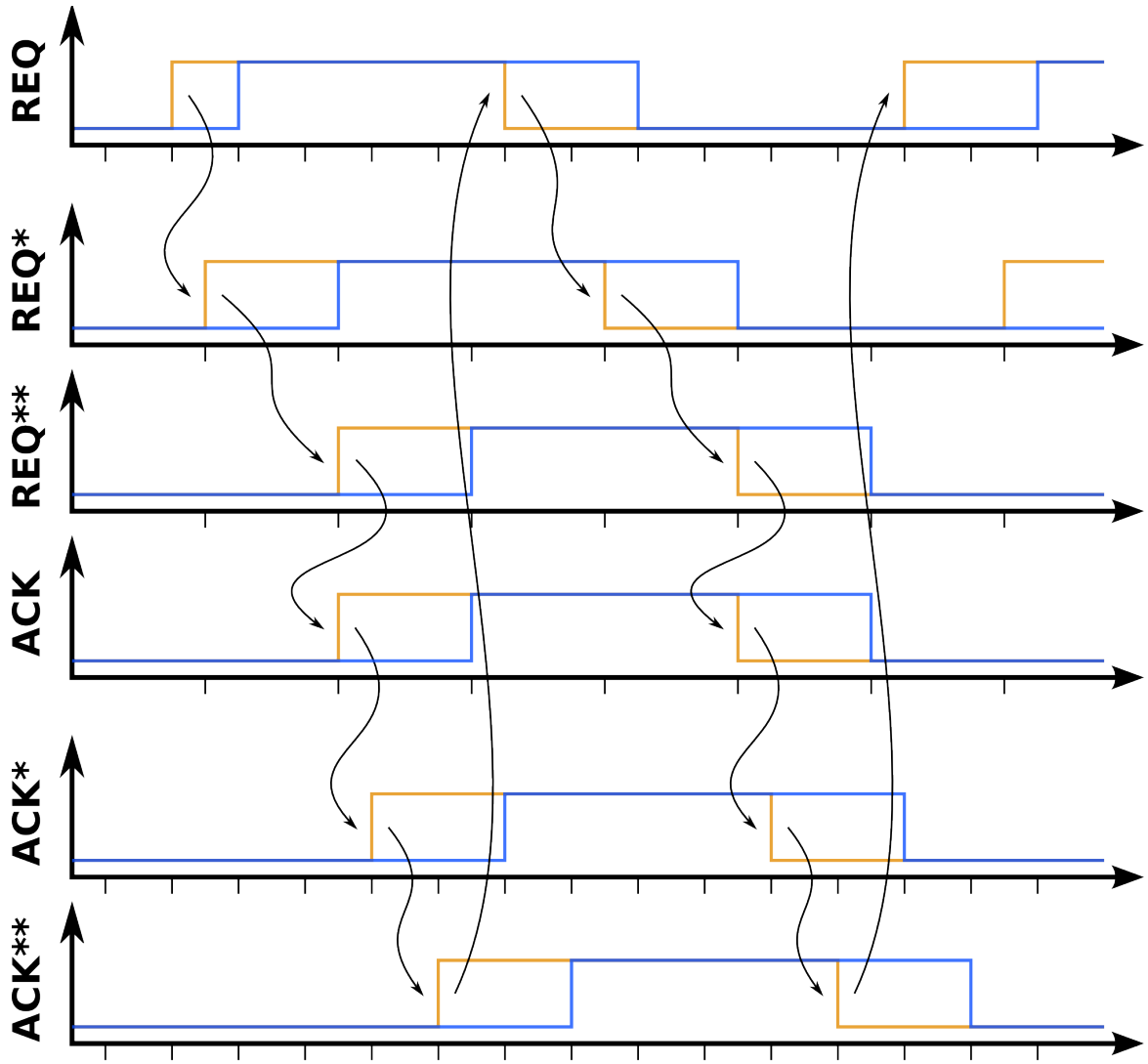

Figure 4.12: REQ/ACK-protocol for flag transfer

CD_a two synchronizers are needed. The signals after the first resp. second register of the particular two-stage synchronizer are tagged with * resp. ** each.
This approach guarantees not only that CD_b has received the flag but also that CD_a waits long enough between two subsequent flag-pulses. The obvious downside of this scheme is the long time between two rising edges on ACK that is introduced by the double cross-CD synchronisation. Depending on the exact relationship between the two clocks the runtime of the protocol may vary (orange vs. blue waveform). Figure 4.13 in the following section depicts a circuit realizing the protocol.

This leads to

Rule 2: To ensure the reception in the destination CD a signal has to be asserted at least one clock period of the destination CD or a REQ/ACK-protocol has to be deployed.

More advanced scenarios and possible solutions for synchronizing a single-bit signal can be found in [26], chapter 9.0.

4.2.3.3 Pitfall 2: Synchronization of a bus

A problem similar to the race condition presented above emerges when multiple synchronizers are used imprudently to transfer bus signals between CDs. Assume each single bit of the bus is connected to one dedicated synchronizer. After reconsidering that the respective delay of each of these synchronizers can only be specified with a some uncertainty the problem becomes visible: If the data on the bus changes close to the active clock edge of the receiving system it might happen that

- some synchronizers update their output to the value of their associated bit after the transition (short delay), and that
- at the same time the other synchronizers will miss the transition and therefore still output the previous value for one more clock cycle (long delay).

This might occur because

- some bit lanes are slightly shorter than others and the new (bit-)values therefore arrive earlier at their respective synchronizer than other ones, or because
- some of the synchronizers sample earlier than the others, e.g. because of skew on the clock network.

Therefore, some bits of the data word will be updated and others won't. This will result in a partially updated word similar to the situation described in section 3.3.2 (`xdio`-module). The fault described above is a race condition between many bit lanes crossing over to CD_b (with distinct delays introduced by the synchronizers which themselves depend on the exact timing of the respective bit lane). This fault will be called a multi-bit synchronization failure and again is not related to metastability.

If no knowledge about the data on the bus is available the only safe way to transfer a bus is to make use of the REQ/ACK-protocol introduced in section 4.2.3.2: In the beginning the sending side (CD_a) latches the data into its output buffers and starts driving the bus `X`. At the same clock cycle it asserts REQ which is used to inform the receiving side that valid data is available ("request to fetch data"). The receiving side (CD_b) will latch the data on bus `X` after receiving REQ. After that, it will assert ACK to inform the sending side of the successful transfer. The sending side must continue driving bus `X` (with the same word) until it has received the acknowledge from the receiving side. After that, the process may start from the beginning. Figure 4.13 shows one possible realization.

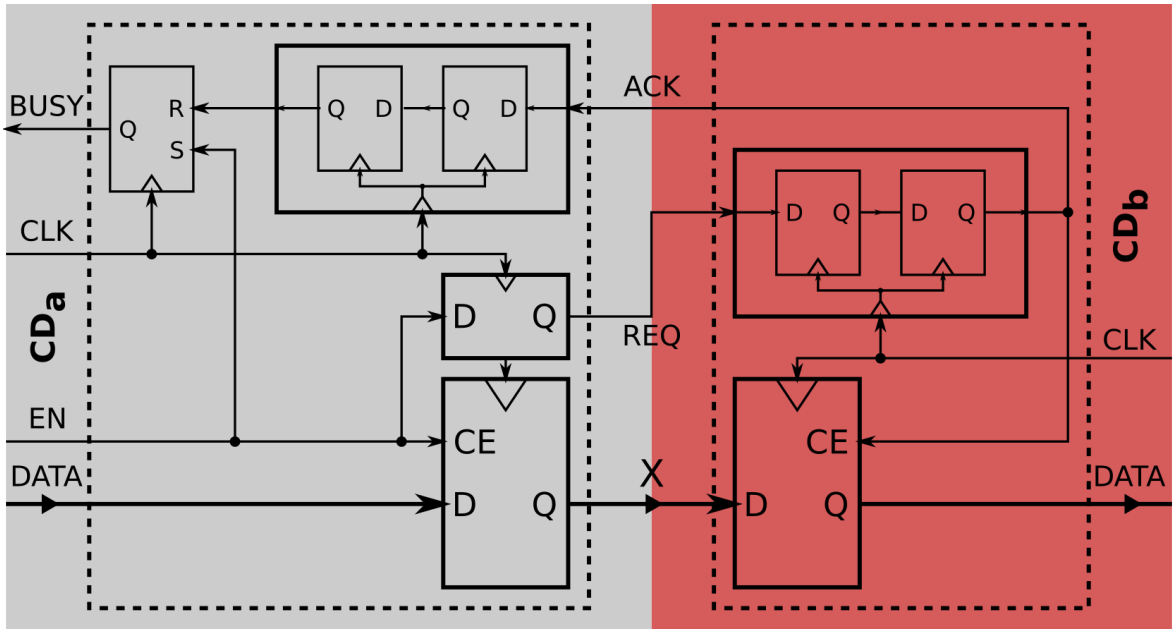

Figure 4.13: REQ/ACK-protocol used to transfer a bus

In contrast to the scenarios described earlier it is perfectly acceptable to use a multi-bit register in CD_b to store the data on bus `X` even its inputs are driven by CD_a. The reason therefore is that because of the REQ/ACK-protocol the receiving side knows for sure that the bus signals are stable when REQ** is asserted.
This method of transferring a bus inherits the disadvantage from the REQ/ACK-protocol mentioned before: The two synchronizers involved in the data flow control lead to a long runtime of the protocol which limits the number of bus-beats per time and therefore the bus throughput.

The following presents an exemplary special case scenario where some knowledge about the data on the bus is available: The value of a 3 bit binary counter has to be transferred from CD_a (slow) to CD_b (fast). The internal structure of the logic in CD_a guarantees that the counter either only increments by one (per CD_a-clock cycle) or holds its current value. To visualize the impact of an improper CDC using multiple synchronizers at first the worst-case outcome will be studied for this case. After that, an alternative approach that makes use of the knowledge about the data will be presented.

Table 4.1 shows all possible decimal values (x_{dec}) and their binary equivalent (x_{bin}). The number of changed bits between two values (when increasing x) is given in the third row (n_{bin}) and the respective bits are underlined in row x_{bin}.

If the 3 bit bus carrying the binary counter value x_{bin} would be transferred to CD_b using three synchronizers a multi-bit synchronization failure could occur. When the counter value x increases all or only some of the changed bits (marked in row x_{bin}) could be delayed (by one clock cycle of the destination CD). Row w_{bin} shows the worst-case result for each increment. Delayed, unchanged bits are overlined while successfully transferred

x_{dec}	0		1		2		3		4		5		6		7		0
x_{bin}	000		00$\underline{1}$		0$\underline{10}$		0$\underline{11}$		$\underline{100}$		10$\underline{1}$		1$\underline{10}$		11$\underline{1}$		$\underline{000}$
n_{bin}		1		2		1		3		1		2		1		3	
w_{bin}			00$\overline{0}$		0$\overline{0}\underline{0}$		01$\overline{0}$		$\overline{0}\underline{00}$		10$\overline{0}$		1$\overline{0}\underline{0}$		11$\overline{0}$		$\overline{111}$
w_{dec}			0		0		2		0		4		4		6		7
δ_{bin}			1		2		1		4		1		2		1		7

Table 4.1: Synchronization errors for binary up-counter

ones are underlined. The decimal equivalent of w_{bin} is given in the next row (w_{dec}) while the last row specifies the resulting error $\delta_{bin} = |x - w|$.

Example: When x changes from 1 (001) to 2 (010) the two rightmost bits change and are therefore susceptible to being delayed by the synchronizers. If the middle bit gets delayed (and therefore stays 0) but the right bit (the LSB) gets successfully updated to 0 the resulting bit sequence will be 000 (0). The difference between the expected value $x = 2$ and the received value $w = 0$ is $\delta_{bin} = 2$.

It can be seen that the error δ_{bin} gets as large as 7 which is the maximum span covered by the counter. This means that (using this approach) the receiving side is unable to recover the actual value because it was transferred using binary coding and might have been damaged (and therefore invalidated) by the synchronization to the target CD.

Apart from transferring the 3 bit binary value using the REQ/ACK-scheme a more time-efficient approach can be deployed here because it is known that the value on the bus is generated by a counter that either keeps or increases the value by one.

To circumvent the problem of a multi-bit synchronization failure the binary counter value can be converted using a coding scheme that ensures that two adjacent numbers (represented by bit patterns) are only distinguished by one bit. This scheme is called Gray-Code named after its inventor Frank Gray who patented it in 1953. The patent specification can be found in [30].
Because the counter in this special case scenario only increases by one (per clock cycle of the sending $\mathrm{CD_a}$) it can be guaranteed that only one bit of the Gray-encoded value on the bus might have changed when the bus is sampled in $\mathrm{CD_b}$ (which is faster than $\mathrm{CD_a}$). Table 4.2 again shows all possible counter values x in decimal (x_{dec}) and their equivalent in Gray-coding ($x_{Gr.}$). The number of changed bits between two values (when increasing x) is given in the third row ($n_{Gr.}$) and the respective bits are underlined in row $x_{Gr.}$. It can be seen that because of the Gray-coding $n_{Gr.}$ is equal to 1 for all possible counter transitions (because there is only one changed bit between two adjacent numbers).

If the 3 bit bus carrying the Gray-encoded value is transferred to $\mathrm{CD_b}$ using three synchronizers no multi-bit synchronization error can occur because only one bit of $x_{Gr.}$ changes when the counter increments. The change of this single bit can either be successfully transferred to $\mathrm{CD_b}$ (which will lead to an up-to-date counter value in $\mathrm{CD_b}$) or be delayed by one clock cycle. The latter is the only possible failure scenario when synchronizing a

x_{dec}	0		1		2		3		4		5		6		7		0
$x_{Gr.}$	000		00$\underline{1}$		0$\underline{1}$1		0$\underline{1}$0		$\underline{1}$10		11$\underline{1}$		1$\underline{0}$1		10$\underline{0}$		$\underline{0}$00
$n_{Gr.}$		1		1		1		1		1		1		1		1	
$w_{Gr.}$			00$\overline{0}$		0$\overline{0}$1		01$\overline{1}$		$\overline{0}$10		11$\overline{0}$		1$\overline{1}$1		10$\overline{1}$		$\overline{1}$00
w_{dec}			0		1		2		3		4		5		6		7
$\delta_{Gr.}$			1		1		1		1		1		1		1		1
δ_{bin}			1		2		1		4		1		2		1		7

Table 4.2: Synchronization errors for up-counter represented in Gray-coding

Gray-encoded (multi-bit) signal that only changes by one per clock cycle (of the sending CD). Because this is the only possible failure this case is the worst-case scenario.
When the change of this single bit is delayed the output of the three synchronizers will stay the same (and $w_{Gr.}$ therefore will be identical to the value of $x_{Gr.}$ before the counter has incremented). Because the receiving CD is faster than the sending CD the next synchronization attempt will take place before the counter in $\mathrm{CD_a}$ increments its value for a second time. This synchronization attempt will (most likely) be successful (because the bus has been stable for approximately one clock period) and therefore the new, updated value will appear at the outputs of the synchronizers.

It can be seen that the worst-case error $\delta_{Gr.}$ is always equal to 1 when Gray-coding is used[9)]. This small difference will only be present for one clock cycle of the receiving CD ($\mathrm{CD_b}$). After the next active clock edge the correct value will be available.

In others words: The counter value received in $\mathrm{CD_b}$ can always be used without concerns for its validity. In case of a clock event (in $\mathrm{CD_b}$) close to a counter transition the CD crossing might take an additional clock cycle, therefore the received counter value might be off by one. This situation (depicted in figure 4.14) is perfectly legal because the counter value just changed when it was read by $\mathrm{CD_b}$ so both the old and the new value are considered correct. The obvious advantage of this approach is the much higher possible data rate compared to the REQ/ACK-protocol because there is no need to wait for an acknowledge from the receiving side.

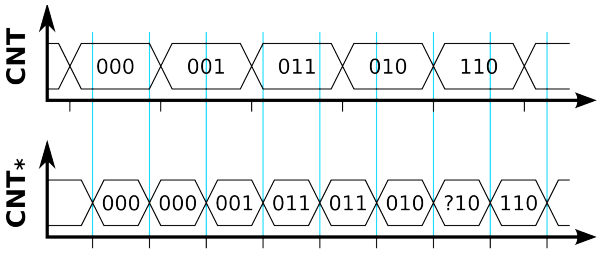

Figure 4.14: Counter synchronization using Gray-code (slow to fast)

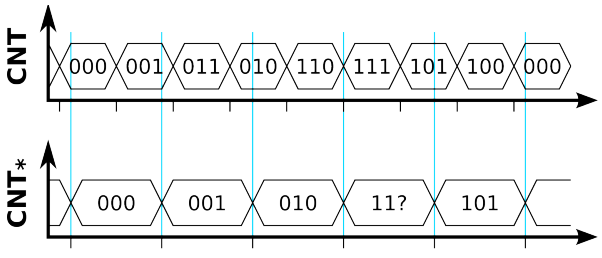

Figure 4.15: Counter synchronization using Gray-code (fast to slow)

[9)] For reference the worst-case error found for binary coding (δ_{bin}) is given in the last row.

This approach will also work if CD_a is faster than CD_b (figure 4.15). The only difference is that in that case the counter in CD_a might increment by more than one before its (Gray-encoded) value gets synchronized into CD_b. The Gray-coding guarantees that only one bit lane on the 3 bit bus will change after the active clock edge of the sending CD (CD_a). The minimum interval between two single-bit changes on the bus is therefore equal to the clock period of CD_a. Because this clock period will be (by magnitudes) longer than the metastability window[10)] of the synchronizing registers not more than one synchronizer (at one time) might see a transition on its input signal close to the active clock edge (of CD_b).

All the above leads to

Rule 3: If an unknown multi-bit signal has to be transferred from one CD to another the only permissible approach is the usage of the REQ/ACK-protocol. If exact knowledge about the multi-bit signal is available special techniques (e.g. Gray-coding) might allow the successful synchronization using multiple single-bit synchronizers.

4.2.3.4 Simulation issues

Please note that synchronization errors related to metastable events are hard to hit in simulation and already should be handled during the design phase of the system.

Usual register simulation models just verify whether setup- and hold-times have been met. If these timing requirements have not been met the model will drive the output of the register to `'X'` (unknown). The output will remain unknown until a following active clock edge (with proper input timing) brings the register back to a defined state. Downstream logic as well as downstream registers will propagate this (undefined) value into the design and may cause a lot of other signals getting undefined as well. This behaviour is only observed in simulation and does not reflect the physical reality inside a chip because an "unknown voltage level" does not exist. These unknown signals are usually a good indication that the design violates the timing requirements for synchronous circuits (section 4.2.2.1). But they also make the simulation of asynchronous interfaces (where timing violations are expected to happen) impossible.

In a proper (synchronous) FPGA design the only location where timing violations are expected and tolerated are all the registers inside the synchronizers.
To transfer this situation into simulation `'X'`-propagation has to be disabled for the first register of each synchronizer. While timing violations in reality could also occur in the second register this will never be observed in simulation. The reason therefore is that Xilinx chose that if a timing violation occurs at a register with disabled `'X'`-propagation the register will just continue driving the previous value to its output. Hence the timing requirements of the subsequent register(s) will never be violated.
The Xilinx software offers multiple ways to disable `'X'`-propagation on single registers using the `ASYNC_REG` constraint. More information can be found on page 87 of [17].

[10)] The blue lines marking the active clock edges of the receiving CD also can be interpreted as (exaggerated) metastability windows of the receiving registers.

4.2.3.5 Summary

This section described a classical synchronization circuit, the two-stage synchronizer. It is capable of increasing the mean time between two metastable events injected into the system by magnitudes. The downside is that it causes an uncertain amount of latency that can not be avoided. This variable latency leads to restrictions concerning the transfer of single- and multi-bit signals between different CDs.

Single-bit control signals can safely be transferred if the asynchronous signal is only synchronized by one synchronizer. Additional care is needed to ensure the safe reception of short pulses if the sending system is faster than the receiving system. If the frequency relation between the systems is unknown a REQ/ACK-protocol with additional latency has to be deployed.

Multi-bit signals (buses) generally have to be transferred by holding the data until the receiving side has acknowledged the reception. The REQ/ACK-protocol already mentioned may be reused therefore. Only if additional knowledge about the data on the bus is available special coding schemes may allow the transfer without such a protocol.

The techniques describes above are not only used to interface integrated circuits but also are essential to implement the independent clock FIFO described in the following section.

4.2.4 Clock domain crossing with the independent clock FIFO

Regular synchronous FIFOs usually feature a write- and a read-port that both are synchronous to a single clock input. Various status lines as e.g. FULL (at the write-port) or EMPTY (at the read-port) are generated by the FIFO and have to be evaluated by the surrounding logic before accessing the FIFO to avoid over- or underflows.
These flags can be created easily, e.g. using a single binary counter that holds the so called FIFO fill level (the number of words currently stored in the FIFO). A write access increments, a read access decrements the counter. In case of no accesses or simultaneous read- and writes the counter keeps its value. The flags then simply can be generated by comparing the counter to zero (for EMPTY) resp. to the FIFO depth (for FULL). Another possibility to determine the FIFO fill level is to compare the write- and read-counters that point to the memory locations which will be written resp. read next.
The actual data can be stored in any kind of (synchronous) memory that is fast enough for the targeted application. If a single port memory shall be used the FIFO core and the memory itself have to be clocked at a frequency at least three[11)] times faster than the clock of the surrounding logic.

Independent clock FIFOs however have to be designed to achieve the same functionality with their write- and read-ports being in different clock domains, respectively. This complicates not only the actual data storage but also particularly the generation of the

[11)] This allows the core to fetch (two clocks) and to write (one clock) new data – if required – in one clock cycle of the surrounding logic.

status lines.
In a Xilinx FPGA the data storage can be realized using (hard core) dual-port Block-RAM (see [21], page 151 ff, for more details) that itself supports accesses from two clock domains. The actual challenge is the design of the accompanying logic that contains read- and write-pointers, creates the status signals fast enough and also takes care of transferring status information between both domains. The latter is needed because both EMPTY (read clock domain) and FULL (write clock domain) are depending not only on the value of the counter of their respective clock domain but also on the value of the counter located in the other clock domain. The reason therefore is that the status signals depend on the FIFO fill level which only can be determined when both write- and read-pointer are known. Therefore, the read-pointer has to be transferred to the write clock domain to generate FULL and the write-pointer has to be transferred to the read clock domain to create EMPTY.

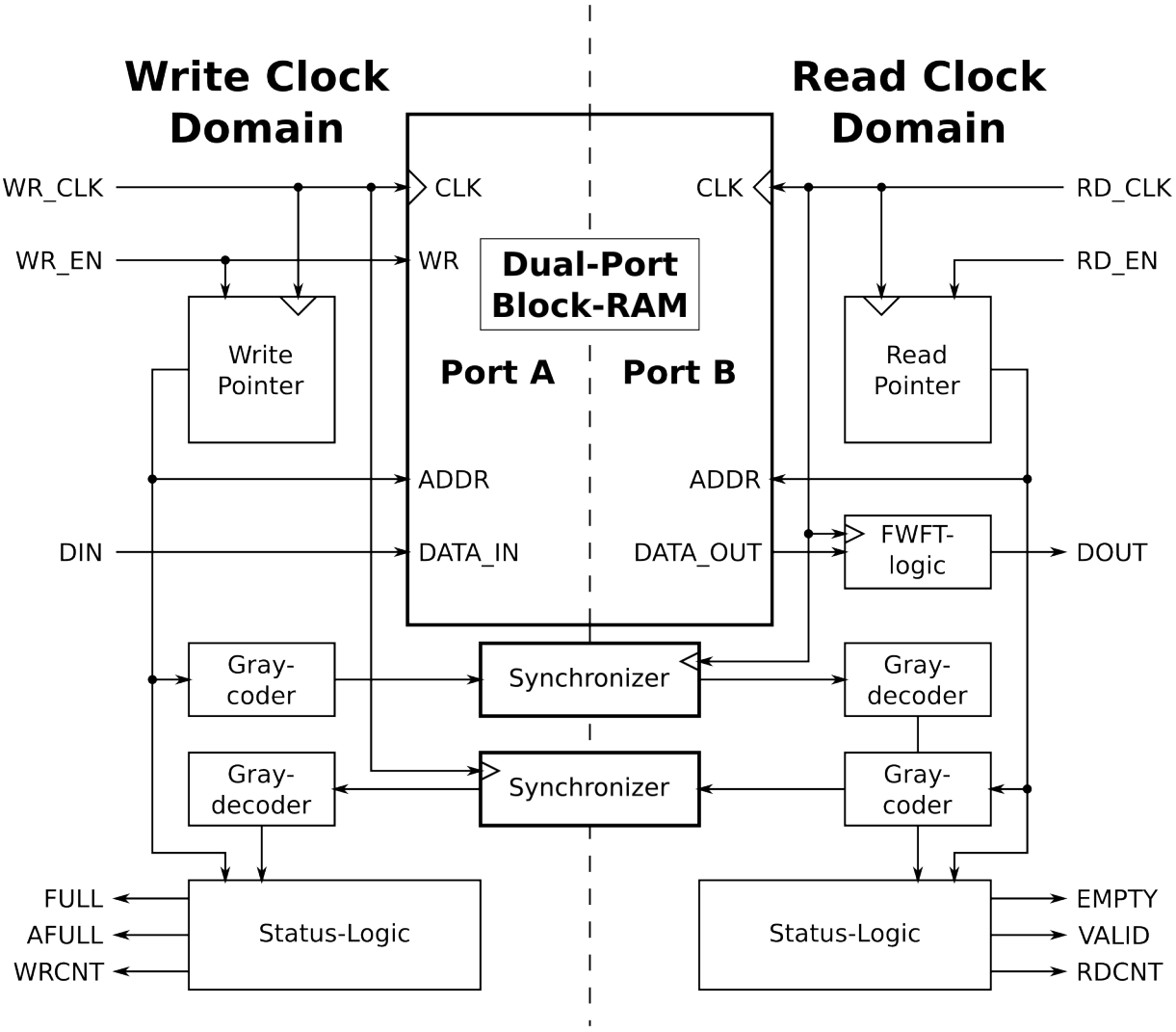

Figure 4.16: Block diagram of a FIFO with independent clocks

Figure 4.16 shows a design of such a FIFO based on information taken from the Xilinx documentation. At first sight a strict division in a write- and a read-CD is evident. All the signals entering and leaving the FIFO in the write-CD form the so called write-port of the FIFO, all the incoming and outgoing signals of the read-CD form the read-port.

All signals of one particular port are synchronous to the respective clock-input of the port. This means that all signals connected to the inputs of the particular port have to comply with setup- and hold-timing requirements in respect to their associated clock and that all outputs will change after a clock event on the associated clock.
It is a vital requirement that the port signals are only used in their own clock domain, because otherwise an illegal CDC would be created. The only elements crossing the CDs are the dual-port Block-RAM and the two synchronizers inside the FIFO.

Both write- and read-address inputs of the dual-port Block-RAM are driven by a counter ("write- resp. read-pointer") synchronous to the respective domain. These pointers work in exactly the same way as they do in a synchronous FIFO: The write-pointer holds the address that will be written next and the read-pointer holds the address that will be read next. If a word was written the write-counter will be increased by one, if a word was retrieved the read-counter will increment by one.

To generate the essential status lines FULL and EMPTY both counters have to be available in both CDs. The read-port's status logic will assert EMPTY if read- and write-counter are equal whereas the write-port's status logic will assert FULL if the difference of the two counters, the FIFO fill level, is equal to the FIFO depth. Therefore, the read-pointer has to be transferred to the write domain and the write-pointer has to be transferred to the read domain. Both of them are multi-bit signals so all the findings of section 4.2.3 have to be taken into consideration. Because both counters (in one clock cycle of their respective CD) only increase by one (or stay unchanged) the second approach presented in section 4.2.3.3 can be used: By converting both counters into Gray-coding before transferring them into the other CD it can be ensured that the received values always are valid. In case of a synchronization problem because of a counter transition too close to a clock edge of the receiving side the transfer of the new value will take one additional clock cycle.
It has to be verified that the (variable) delay introduced by the synchronizers does not lead to incorrect flags (which then would lead to over- or underflows).

The EMPTY flag is asserted if the read pointer had been incremented that much that it has reached the same value as the write pointer. The write pointer seen by the read-CD may be smaller than the actual value of the "real" write pointer in the write-CD because of the delay introduced by the CDC. It will never be bigger because the write-pointer only increments, not decrements[12)].
The important edge of EMPTY is the rising one that informs the surrounding logic that no more data is available in the FIFO. If this edge is generated too late an underflow will occur (because the logic would try to read from an empty FIFO).

[12)] If it would decrement the decremented value might be delayed by the CDC which would lead to a too large value seen by the read-CD.

If the write-pointer seen by the read-CD is up-to-date the EMPTY flag will be asserted exactly after the last word has been read. If the write-pointer seen by the read-CD is delayed (and therefore smaller than the "real" one) EMPTY will be asserted too even though data would still be available in the Block-RAM. Therefore, no underflow will occur and the delay introduced by the CDC of the write-pointer does not pose a problem to the timely assertion of EMPTY. After the updated value of the write-pointer has reached the read-CD EMPTY will be deasserted again and the surrounding logic will continue reading from the FIFO (until it is empty again).

In other words: The delay introduced by the CDC of the write-pointer might only delay the unimportant falling edge of EMPTY, the important rising edge will always be generated right on time.

The FULL flag is asserted if the write pointer had been incremented that much that the FIFO fill level has reached the FIFO depth. The read pointer seen by the write-CD may be smaller than the actual value of the "real" read pointer in the read-CD because of the delay introduced by the CDC. It will never be bigger because the read-pointer only increments, not decrements[13)].
The important edge of FULL is the rising one that informs the surrounding logic that no more data may be written to the FIFO. If this edge is generated too late an overflow will occur (because the logic would try to write to a FIFO that can not store one more word).
If the read-pointer seen by the write-CD is up-to-date the FULL flag will be asserted exactly after the FIFO fill level has reached the FIFO depth. If the read-pointer seen by the write-CD is delayed (and therefore smaller than the "real" one) FULL will be asserted too even though some more data could be stored in the Block-RAM. Therefore, no overflow will occur and the delay introduced by the CDC of the read-pointer does not pose a problem to the timely assertion of FULL. After the updated value of the read-pointer has reached the write-CD FULL will be deasserted again and the surrounding logic will continue writing to the FIFO (until it is full again).

In other words: The delay introduced by the CDC of the read-pointer might only delay the unimportant falling edge of FULL, the important rising edge will always be generated right on time.
This situation is symmetrical to the situation described for the EMPTY flag.

Compared to transferring the counter values using the REQ/ACK-protocol introduced in section 4.2.3.3 the deployment of the Gray-code-based approach leads to a higher FIFO throughput. The reason therefore is that the additional latency of the REQ/ACK-protocol would slow down the updating of both counters which would lead to the unneeded assertion of both EMPTY and FULL.

This topic has been covered by many publications: [27] gives a good introduction to FIFOs covering pointer generation as well as problems that come with creating an independent

[13)] If it would decrement the decremented value might be delayed by the CDC which would lead to a too large value seen by the write-CD.

clock FIFO. [28] was co-authored by a Xilinx employee and presents a sophisticated approach: Instead of synchronizing Gray-coded counter values to the opposite clock domain an asynchronous compare-unit is used to create the status flags. This topic also served as starting point for numerous discussions in the Usenet, e.g. [14] or [12] and related messages.

The result of all the efforts above is a fast independent clock FIFO that does not impose any restrictions on the two clock signals. As long as the FIFO is not full resp. empty one word can be written resp. read in every clock cycle of the associated CD. Not only both clock frequencies but also the phase relationship between the two active clock edges may be arbitrary. Please note that both clocks have to be continuous free running clocks because they drive, among others, the two synchronizers used to transfer the counter values between the CDs. If the clocks stop between accesses the status flags (e.g. FULL and EMPTY) might not get updated because they depend on these counter values being updated continuously.

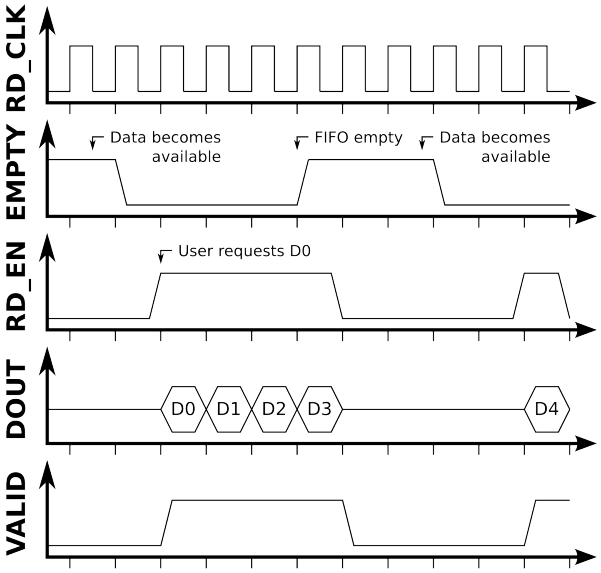

Figure 4.17: FIFO read (standard)

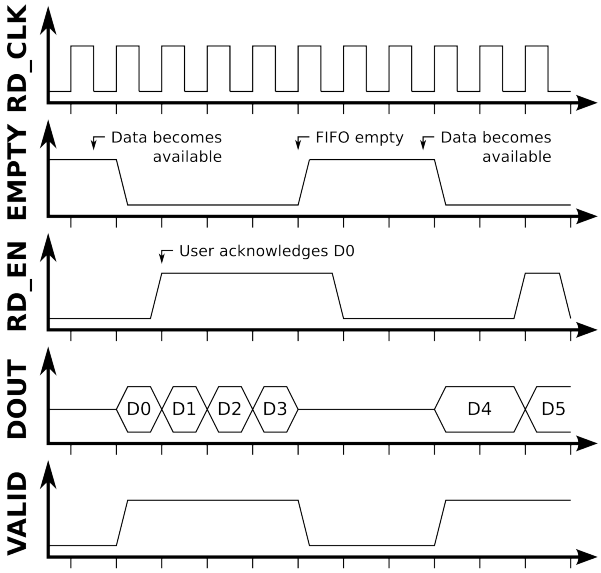

Figure 4.18: FIFO read (FWFT)

The Xilinx software comes with a versatile tool that allows the creation of heavily customized FIFO macros (soft cores), the "FIFO core generator" (see [19]). Some of the core's design options and features are listed below:

- The core can be configured to use various types of memory for data storage: Block-RAM is appropriate for large FIFOs while distributed RAM (implemented in the CLBs) may be used for small, localized buffers.
- Both regular synchronous and asynchronous FIFOs with two independently clocked ports may be generated. The core handles all required synchronization between the two CDs.
- Two readout modes are available: A "standard mode" in which each word has to

be requested one clock cycle before it can be read and an alternative mode called "first-word fall-through" (FWFT). In this mode the next word available for readout will "fall through" the FIFO and can be processed immediately. The validity of the data output will be indicated by an additional status line called VALID. After the surrounding logic has processed the data it asserts RD_EN which informs the FIFO to output the subsequent data word (if available). The activation of the FWFT-mode leads to additional logic on the read-port (see figure 4.16). Comparing the read-operations shown in figure 4.17 (standard mode) and figure 4.18 (FWFT-mode) the difference is evident.

- Various data widths and FIFO depths can be selected. If the data storage is implemented using Block-RAM even different input and output data widths can be chosen by the user.
- Various additional status flags as e.g. almost full, almost empty, write acknowledge, (write) overflow, (read data) valid, (read) underflow and programmable full and empty can be generated by the core.
- An (in some cases obligatory) asynchronous or synchronous reset input can be added.
- Finally, both write and read counter values can be output by the core.

4.2.5 Helpful VLIO quirks

The introduction to VLIO given in section 3.3.1 intentionally makes no mention of two characteristics of the VLIO bus protocol because they were not used in the old design. First of all, the RDY-pin (wired in latter revisions of the EyeBot M6-PCB, see figure 3.1) can be used to slow down bus transactions on demand. Second the VLIO-implementation of the PXA255-CPU is capable of burst transactions that not only accelerate the data transfer but also can be used to compensate the lack of a clock signal from the CPU. Both features will be introduced in the following. Based thereon, two distinct approaches to interface CPU and FPGA will be deduced.

4.2.5.1 Using the RDY-pin to throttle the data transfer

If an external device connected to the VLIO-interface needs additional time to handle the current transfer it may use the RDY-pin to advise the CPU to lengthen the current bus cycle. It RDY is deasserted during a VLIO-beat ($\overline{\text{OE}}$ or $\overline{\text{PWE}}$ asserted) the CPU will stop until RDY is asserted again. The RDY signal is routed through a two-stage synchronizer in the PXA-CPU that drives the internal signal RDY_sync and adds a delay of two MEMCLKs. $\overline{\text{OE}}$ or $\overline{\text{PWE}}$ will be asserted for two more MEMCLKs after RDY_sync has been asserted (driven high). Because RDY_sync is the delayed version of RDY this is equivalent to: $\overline{\text{OE}}$ or $\overline{\text{PWE}}$ will be asserted for four more MEMCLKs after RDY has been asserted (driven high). This means that the state of the (external) RDY signal at the

assertion (falling edge) of $\overline{\text{OE}}$ or $\overline{\text{PWE}}$ determines whether any wait states will be inserted or not because their regular assertion time (without wait states) is equal to 4 MEMCLKs.

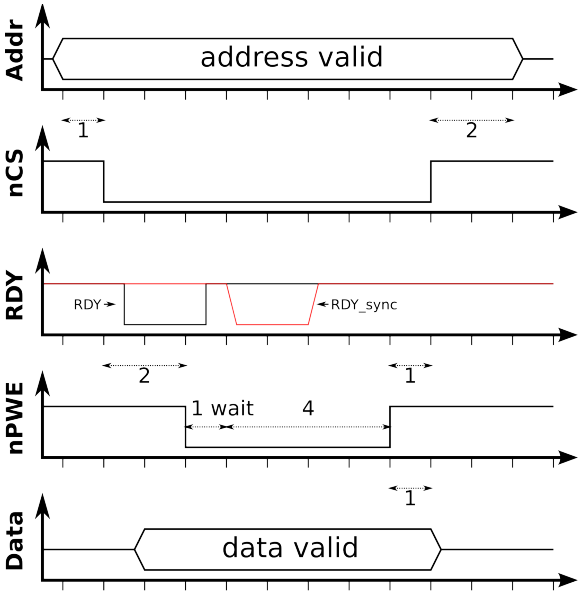

Figure 4.19: VLIO write with wait states (one additional MEMCLK)

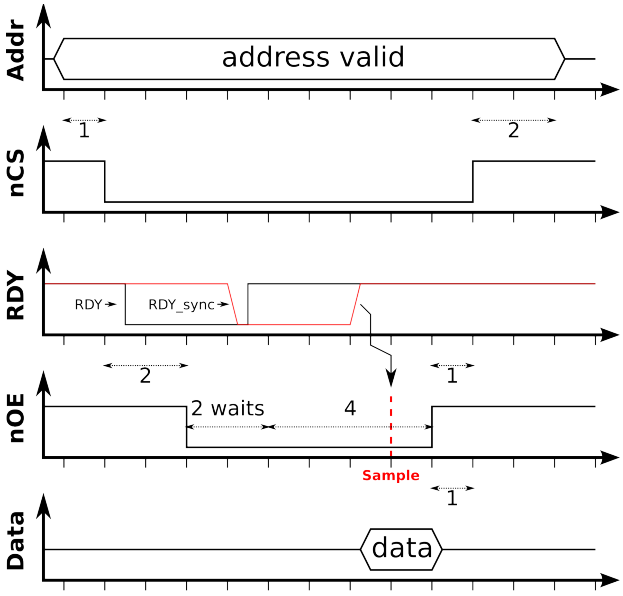

Figure 4.20: VLIO read with wait states (two additional MEMCLKs)

Figure 4.19 shows an example for a VLIO write access: The FPGA deasserts RDY after it has been selected by the CPU. After the CPU asserted $\overline{\text{PWE}}$ the FPGA asserts RDY because the FPGA is always fast enough to handle write-requests. This late assertion (after $\overline{\text{PWE}}$ has already been asserted) will lead to one wait state. This additional MEMCLK can not be avoided if read-accesses have to be throttled because the direction of the transfer (write or read) is unknown to the FPGA until $\overline{\text{OE}}$ or $\overline{\text{PWE}}$ is asserted[14]. The RDY signal will then be synchronized into the CPU-CD which will take two MEMCLKs. Two MEMCLKs after RDY_sync has been asserted $\overline{\text{PWE}}$ will be deasserted and the transaction will end as usual.

Figure 4.20 shows an example for a VLIO read access: The FPGA again deasserts RDY after its $\overline{\text{CS}}$ has been asserted. The assertion of $\overline{\text{OE}}$ causes the bus interface of the FPGA to send a read-request to the addressed internal module. The exact time required to handle the read-request might depend on the type of the addressed storage inside the FPGA (register, Block-RAM, etc.), the interconnect delays and on the current phase alignment between CPU- and FPGA-clock. After (a little less than) two MEMCLKs the FPGA has fetched the requested data and asserts RDY. On the first rising edge of MEMCLK after RDY_sync was asserted by the synchronizer the data bus will be sampled by the CPU. One MEMCLK later $\overline{\text{OE}}$ will be deasserted again and the transaction will end as usual.

[14] If RD/$\overline{\text{WR}}$ would be connected to the FPGA this wait state for write-accesses could be avoided.

Looking back at figure 4.12 (section 4.2.3.3) some similarities between the REQ/ACK-protocol and the operating mode of the RDY-pin can be identified. The equivalent signal names of the REQ/ACK-protocol are given in brackets.
For write-accesses (data transfer from CPU to FPGA initiated by CPU) the VLIO-protocol is identical to the REQ/ACK-scheme. The CPU is equivalent to CD_a (asserting REQ to signal valid data) while the FPGA represents CD_b (asserting ACK). $\overline{PWE}$ (REQ) informs the FPGA that data from the CPU is available on the data bus `X` and the FPGA acknowledges the reception by asserting RDY (ACK).
For read-accesses (data transfer from FPGA to CPU initiated by CPU) the handshaking-scheme is extended to a three-way-handshake because the direction of the data flow is reversed: The FPGA now is equivalent to CD_a (asserting REQ to signal valid data) while the CPU represents CD_b (asserting ACK). $\overline{OE}$ (no equivalent in the two-way REQ/ACK-protocol) informs the FPGA that the CPU wants to read data. The FPGA then fetches the requested data, drives the data bus and asserts RDY (REQ) to inform the CPU that the data bus now is stable and may be latched. After the CPU has latched the data on the bus it deasserts $\overline{OE}$ (which is equivalent to the assertion of ACK) to inform the FPGA of the successful transfer.
The above leads to

Approach 1: To achieve a reliable data transfer on the asynchronous VLIO-interface between CPU and FPGA a request/acknowledge-protocol can be implemented using the CPU's RDY-pin. The RDY-pin will be used to lengthen the read bus-cycle until the FPGA has fetched the requested data. This allows read-accesses to arbitrary, possibly slower, memory locations as well as arbitrary write-accesses.

4.2.5.2 Using the VLIO burst mode to generate a clock signal

The FIFO-based approach proposed in section 4.2.1.2 made use of the CPU's SRAM interface and could not be implemented because there is no clock signal leaving the gumstix. Its only disadvantage was that read-accesses had to be slowed down because of the (variable) delay introduced by the two FIFOs. Because the configuration of the bus timing affects both read- and write-accesses the latter got slowed down as well.
If (in addition to the VLIO-interface) a CPU clock signal would have been available to the FPGA a slightly modified version of the FIFO-based approach presented in section 4.2.1.2 could have been implemented. This approach would use the RDY-pin of the VLIO-interface and the status flags of the FIFOs to lengthen a read access only by exactly the time that is needed to transfer the requested data back to the CPU-CD. This would lead to a non-degraded write-performance because only read-accesses would be slowed down accordingly. More details on this notional approach can be found in appendix B.
During research it was discovered that the lack of a CPU clock signal does not wholly preclude the deployment of this FIFO-based approach if some trade-offs can be accepted. These trade-offs and their causes will be explained in the following.

Apart from the single-beat transactions already shown in section 3.3.1 (figure 3.2 and figure 3.3) the VLIO interface is also capable of a burst-mode that transfers more than

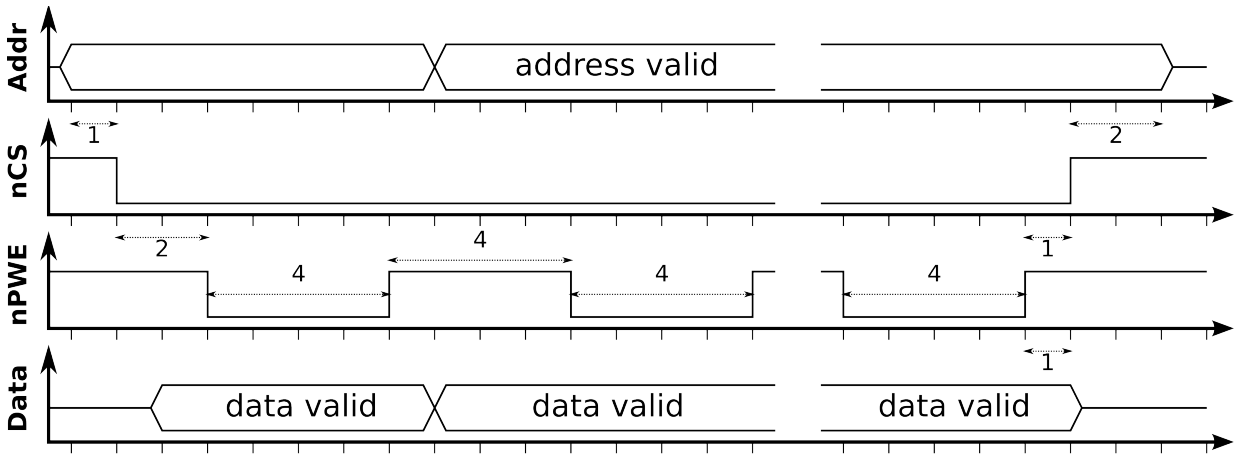

Figure 4.21: VLIO burst write

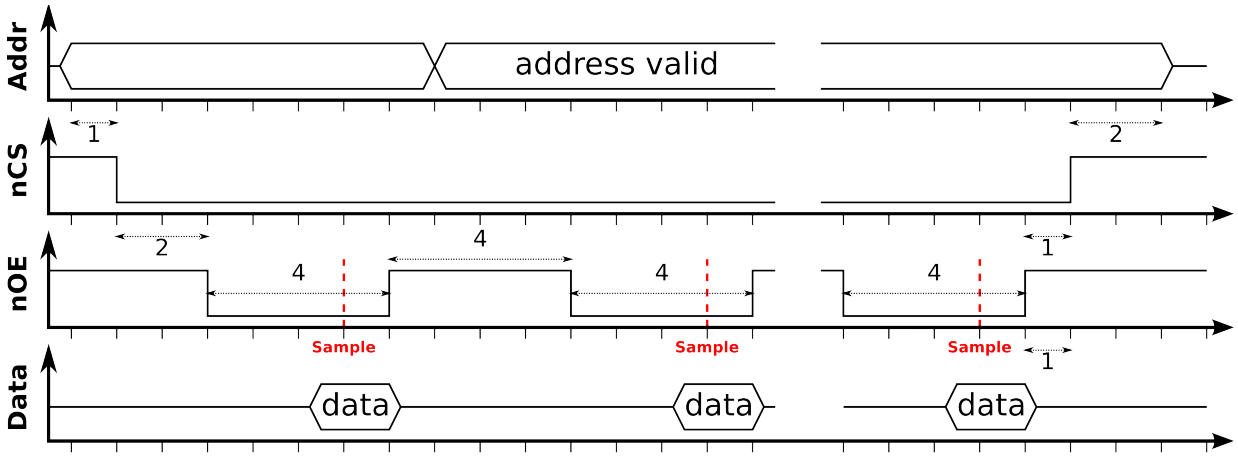

Figure 4.22: VLIO burst read

one beat per transaction. Figure 4.21 shows a burst-write access, figure 4.22 a burst-read access. One property of this transfer-mode is that the control lines $\overline{\text{OE}}$ resp. $\overline{\text{PWE}}$ toggle once per beat and therefore look similar to a clock signal. Because the signals are not toggling all the time they are considered non-continuous clocks.

Looking at appendix B it can be found that non-continuous clocks might lead to a deadlock if the RDY-pin is used for flow control: Because the status flags of the FIFOs are not updated (if a continuous clock is not available) it is impossible to drive RDY in a way that stops the CPU if a read-request is still processed by the FPGA (equivalent to FIFO2 still being empty) or if FIFO1 is full. Because of the former this modified FIFO-based approach can not be used to implement reads from arbitrary memory locations (which could not be answered fast enough). Because of the latter the CPU must not write to FIFO1 except it is somehow ensured that FIFO1 will not overflow. Therefore, some other way to implement the flow control that protects FIFO1 from over- and FIFO2 from underflows has to be found.

This obviously has to be done on software-level because the hardware-based solution using RDY is unavailable. In the event the CPU wants to transfer a certain amount of data to the FPGA it first has to check whether there is enough space in FIFO1 to store this specific number of words. If enough space is available the CPU may write the data because FIFO1 will not overflow.
In the event the CPU wants to read a certain amount of data it first has to verify that enough data is ready in FIFO2. Then the CPU may read that specific number of words because FIFO2 will not underflow.
These checks take a certain time and therefore introduce an overhead to each data transfer. To reduce the impact of this overhead on the achievable data rate only large amounts of data should be transferred. Therefore, the target pursued here is to establish a way to transfer large, contiguous amounts of data with the highest achievable data rate.
For the sake of simplicity it is appropriate to implement these transfer in a way that the particular FIFO is empty after the transfer has finished. As a rule both FIFOs therefore can be assumed to be empty before the start of a transfer.
Because the FIFOs will only be used for large, contiguous amounts of data, but not for accesses to arbitrary memory locations, there is no need to transfer more than one address to the FPGA-CD. The reason therefore is that only the address of the first word is relevant because the subsequent words will (by definition) just be saved to resp. read from the subsequent memory locations. This single address will be called pointer.

Such a transfer will consist of two phases initiated by the CPU:

Setup phase: In case of a write access two steps are needed: First it will be verified whether FIFO1 is actually empty as it should be (by definition). After that, the pointer to the memory area inside the FPGA that shall be written to and the transfer size will be sent to the FPGA.

In case of a read access three steps are necessary: First it has to be verified whether FIFO2 is actually empty as it should be (by definition). After that, the CPU will request the FPGA to fill FIFO2 with the data needed (which again will be identified by a pointer to a memory area inside the FPGA and the size of the transfer). Finally, the CPU has to wait until the FPGA has initiated the transfer to FIFO2 to make sure that enough data is available in the FIFO when the CPU starts reading.

For both accesses the transfer size obviously has to be smaller than the size of the respective FIFO to avoid over- resp. underflows.

Data transfer phase In this phase the actual data will be transferred between CPU and FPGA. Because it has been verified that FIFO1 is empty resp. FIFO2 contains enough data neither an over- nor an underflow will occur. This strategy compensates the lack of a hardware flow control on the VLIO bus.

In other words:
To transfer data from CPU to FPGA the CPU first informs the FPGA of the upcoming transfer and supplies a pointer to the memory area that shall be written to. After that, the CPU writes the data to FIFO1. In the end the FPGA transfers the data from the

FIFO to the memory area specified by the pointer received from the CPU.
To transfer data from FPGA to CPU the CPU first notifies the FPGA to fill FIFO2 with the data needed (which again will be identified by a pointer to a memory area inside the FPGA). After the FIFO has been filled by the FPGA the CPU will receive the data from the FIFO.
These large data transfers and their associated setup phases (to transfer the pointer) will be called FIFO-transactions. They are able to compensate the lack of a hardware flow control and therefore allow a reliable (and fast) data transfer for large, contiguous amounts of data.

Please note that the introduction of these FIFO-transactions is not that a serious drawback (in terms of complexity and overhead) as it might seem at first glance. The reason therefore is that they make use of the VLIO burst mode which is anyway designated for large, continuous data transfers that have to be set up before. Besides that, these high-level transactions ease the arbitration of the SRAM connected to the FPGA. This topic will be covered in section 4.4.1.6.

The above leads to

Approach 2: To achieve high-speed data transfers between CPU and FPGA two signals of the VLIO-interface can be used as (non-continuous) clock signals for the FIFOs. Because the FIFO's status flags are only up-to-date if a continuous clock is available it is impossible to implement a hardware flow control using the RDY-pin in this case. Nevertheless over- and underflows in the FIFOs have to be avoided using some sort of software flow control which leads to FIFO-transactions.

More information on the usage of this interface will be given in the next section (4.2.6), the actual implementation will be presented in section 4.2.7.

Please note that this approach to generate a clock signal for the FIFOs only works if the VLIO-protocol is used. All other interfaces (including the SRAM-interface proposed in [25]) do not provide any sort of signal that marks each single beat of a transfer. Without this knowledge (and without a clock signal from the CPU) the FPGA sooner or later will loose synchronization which again would lead to transfer errors.

4.2.6 Peripheral- and burst-interface

Up to now two distinct approaches to interface CPU and FPGA have been introduced. This section compares the two methods and deduces their respective fields of application on the EyeBot M6. The next section (4.2.7) then presents the actual implementation chosen for the bus bridge in the FPGA.

Section 4.2.5.1 introduced the RDY-pin of the VLIO-interface and concluded with an approach how to implement read- and write-accesses to arbitrary memory locations of the FPGA's address space. Section 4.2.5.2 however presented the VLIO-burst-mode and how it can be used to deploy a fast FIFO-based interface for large amounts of data.

To assess the requirements on the FPGA's bus interface the expected data flow between FPGA and CPU has to be estimated first. This data flow obviously depends on which types of functions are implemented in the FPGA and have to be controlled and utilized by the CPU. Looking back at the block diagram of the EyeBot M6 (figure 2.2) various components connected to the FPGA can be identified. They and their supposed interface requirements are listed in table 4.3.

Component		Direction	Main requirement	
Name	Class	of flow	low latency	high throughput
GPIO	I/O	bidirectional	yes	no
PSDs	Sensor	FPGA → CPU	yes	no
Encoders	Sensor	FPGA → CPU	yes	no
Motors	Actuator	CPU → FPGA	yes	no
Servos	Actuator	CPU → FPGA	yes	no
Camera	Sensor	FPGA → CPU	no	yes
SRAM	Storage	bidirectional	no	yes

Table 4.3: Interface requirements of components connected to the FPGA

Looking at the requirements of the components two distinctive classes of devices can be identified: The first one (mainly used to access components for basic robot control) features mostly single-word accesses (both read and write) and requires a low latency (to facilitate a smooth control of the robot).
The second one features large data transfers between CPU and FPGA and therefore requires the highest possible throughput (to save time that then can be used for computation-intensive tasks on the CPU).

If these two transfer classes are compared with the two available interface methods an obvious conformance can be found: The first approach based on the REQ/ACK-protocol supports both read- and write-accesses to arbitrary memory locations without setup overhead. The second, FIFO-based approach provides high-speed data transfers at the cost of setup overhead. Hence both interface methods are needed on the EyeBot M6 and will therefore be implemented as coexisting subsystems.
Because the two interfacing methods have distinctive focuses it is advisable to use two diverse bus systems inside the FPGA: One for low-latency accesses (via the REQ/ACK-protocol interface) and one for high-throughput transfers (via the FIFO-based interface). These bus systems have been named peripheral- resp. burst-bus, briefly PB and BB[15)].

The following decision was taken during the design phase: The BB will only be used to transfer (large amounts of) data from or to the 18 MBit-SRAM connected to the FPGA.

[15)] The naming was influenced by various factors:
Virtex-based embedded systems usually incorporate a bus system with similar objectives to the PB that is called On-Chip Peripheral Bus (OPB) which lead to the name "peripheral bus".
On the other hand transfers containing image-data will be the most common ones on the BB and hence can be transferred in large chunks (bursts). Therefore, and because the associated communication between CPU and FPGA will be implemented using VLIO-bursts the bus was called "burst bus".

The reason for this decision is that the external SRAM is the sole storage element available to the FPGA that is large enough for image data or other huge data sets (such as a list of feature points generated by an image processing unit in the FPGA).
Because the SRAM will be accessed by multiple units (e.g. a frame grabber storing raw images from the camera(s), the image processing unit already mentioned, the CPU, etc.) the BB has to be multi-master capable.
Because only a limited number of transfer types will be needed it is advisable to implement these transfers as transactions[16)] on the BB. Each transaction consists of an arbitration-, a setup- and a data-transfer phase. Typical examples for the data assigned to a specific transaction type could be one line of a color image, one line of a grayscale image, four specific pixels from a 2x2-window or non-image data such as a list of feature points. Further information on the BB can be found in section 4.4.

Table 4.4 shows a summary of both interfacing methods, their assets and drawbacks, their associated internal bus system, their respective field of application and the assignment to the components resp. modules.

Interface	REQ/ACK-protocol using RDY	FIFO-based (pseudo-clocked)
Description	Uses the RDY-pin of the VLIO-interface to support accesses to arbitrary memory-locations	Uses $\overline{\text{PWE}}$ and $\overline{\text{OE}}$ of the VLIO-interface to generate a clock for the FIFOs
Internal bus	peripheral bus (PB)	burst bus (BB)
Type	single-master system	multi-master system
Facts	managed by CPU per-module enable 10 bit address bus 16 bit data bus arbitrary accesses	managed by SRAM-controller module-based arbitration transfer-type based addressing 16 bit/18 bit data bus transaction-based accesses
Assets	no arbitration and setup needed	high data rate
Drawbacks	low data rate	overhead for transaction setup
Used for	GPIOs, sensors, actuators	SRAM accesses (camera, CPU, etc.)

Table 4.4: Available interface methods and assigned components

As mentioned in section 3.3.2 the old FPGA design already contained an internal bus system (containing asynchronous paths). Because various modules (GPIOs, PWM generation for servo and motor control, data acquisition for encoders and PSDs) were already implemented for this internal bus system the PB was designed to be (mostly) compatible with the old internal bus system. The only feature not (natively) supported by the PB is the generation of a special bus signal called `mod_abit`. It was used to switch the modules into a special mode that allowed the alteration of single bits in a multi-bit register with one single write-access. To recover this functionality the respective inputs of the

[16)] Please note that these transactions are different from the FIFO-transactions defined earlier. The latter ones are used to implement the safe data transfer between CPU and the FIFOs in the FPGA using the VLIO-interface. The transactions on the BB are used to transfer data purely inside the FPGA.

modules were connected to an unused address-line of the PB. This achieved backward compatibility with the old design and therefore with the existing driver library, too.

4.2.7 Design of the bus bridge

The previous section (4.2.6) explained the need for two distinct bus systems inside the FPGA, the peripheral bus and the burst bus. The former will be utilized by the CPU to access various peripheral modules while the latter provides both CPU and internal modules with access to the external SRAM.

Because the interface between CPU and FPGA is asynchronous but FPGAs generally are built to implement synchronous circuits, the asynchronous VLIO bus shall be terminated as "early" as possible inside the FPGA. The requests sent by the CPU using VLIO therefore have to be bridged to the appropriate bus system inside the FPGA (PB or BB). This section is dedicated to the actual implementation of the bus bridge that transfers the asynchronous VLIO bus into the synchronous FPGA.

After the requirements for the bus bridge have been analyzed in the following the implementation of both the REQ/ACK-based (for the PB) and the FIFO-based interface (for the BB) will be presented. Thereafter various measurements of the achievable transfer rate on both interfaces will be shown. The section finally ends with a summary on the bus bridge and a block diagram.

4.2.7.1 Requirements and partitioning

The bus bridge provides the connective link between the CPU and the modules inside the FPGA and therefore is an essential module in the FPGA. It has to implement multiple features:

- process the asynchronous signals from the PXA-CPU and transfer the CPU requests from the CPU-CD to the FPGA-CD,
- decode the address supplied by the CPU and forward the request to the relevant sub-interface (PB, FIFOs or internal control),
- generate the signals on the PB based on the CPU's requests,
- control and monitor the FIFO-ports that are part of the CPU-CD and
- drive the RDY-pin of the CPU's VLIO-interface.

The bus bridge (whose entity is named `bus_bridge`) mainly consists of two main and two supporting subsystems:
A dedicated unit (PB bus bridge) establishes the communication between VLIO and PB while two independent clock FIFOs accomplish the data transfer between VLIO and BB. Both of them implement the CDC between CPU- and FPGA-CD.
Additional combinatorial logic is needed to decode the address supplied by the CPU, to

activate the relevant subsystem and to control multiplexer and tri-state buffer driving the data bus. Furthermore, a control/status-unit for the FIFO-ports that are in the CPU-CD was added.

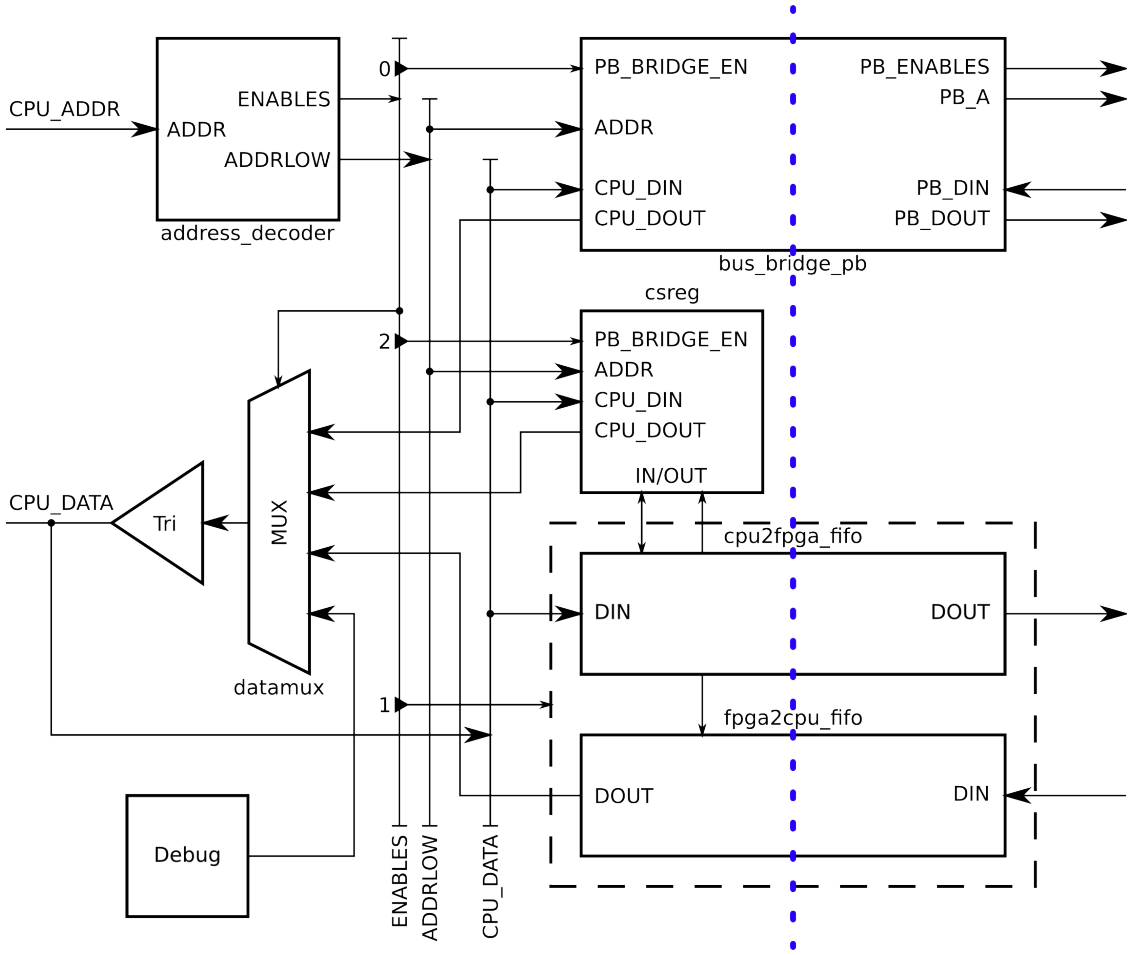

Figure 4.23: Bus bridge (coarse block diagram)

Figure 4.23 shows a coarse block diagram of the bus bridge only depicting address-, enable- and data-signals. All control signals (including RDY) have been omitted for the sake of clarity. The CDC between CPU-CD (left) and FPGA-CD (right) is tagged by the dotted blue line.

The address-decoder decodes the two highest address-bits (`CPU_A[20:19]`), checks whether the FPGA's chip-select (`CPU_NCS`) is asserted and generates a 4 bit-wide enable signal (`ENABLES`). This signal is used to enable one of the three subsystems. Each subsystem is wired to the CPU's data bus (`CPU_DATA`) to receive data during a write-access. If the CPU wants to read from one of these subsystem the data output of the respective subsystem has to be passed through to the CPU's data bus. This is accomplished by a multiplexer that is fed by the outputs of the subsystems and controlled by the enable signal (which determines the active subsystem). The output of the multiplexer (`datamux`) is connected

to a tri-state buffer (controlled by `CPU_NCS` and `CPU_NRD`) that drives the data bus during a read-access.

In the following the subsystems will be presented.

4.2.7.2 Peripheral bus bridge

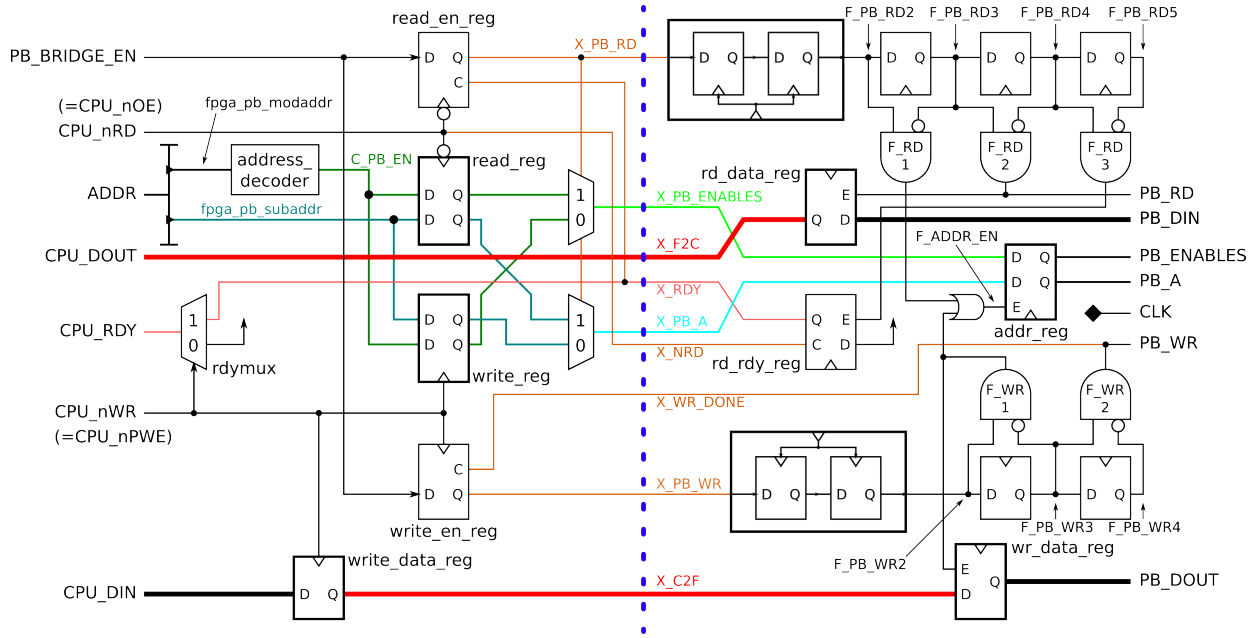

Figure 4.24: PB bus bridge

Figure 4.24 shows the subsystem responsible for the data transfer between VLIO and PB. The respective entity is called `bus_bridge_pb` and is instantiated in `bus_bridge`. Internal signals are named using the following scheme: Signals generated in the CPU-CD are prefixed with `C_` while signals generated in the FPGA-CD are prefixed with `F_`. All signals connecting the two CDs are prefixed with `X_`. On certain registers the asynchronous reset input `C(LR)` is used to bring the unit back to reset state.
For the sake of clarity the clock-net in the FPGA-CD has been omitted (all registers in the FPGA-CD are clocked on the rising edge of `CLK`).

Again the CDC is tagged with a dotted blue line (CPU-CD left, FPGA-CD right). The implementation exactly matches the interfacing requirements of the VLIO-interface and takes care of the CDC between CPU and FPGA. The approach used is similar to the REQ/ACK-protocol introduced in section 4.2.3.3 (figure 4.13). In the following the order of events during write- resp. read-accesses from the CPU will be demonstrated.

Write-accesses in general do not make that high demands on the accessed system's performance compared to read-accesses because only one CDC is needed (from CPU to FPGA). Looking back at figure 3.2 it can be seen that both address- and data-bus are stable for 6 MEMCLKs one MEMCLK after $\overline{CS}$ has been asserted. During this

time the FPGA has to store both address and data. After that, the address has to be evaluated and the data has to be stored to the addressed storage element (register, Block-RAM, etc.). If the RDY-pin of the VLIO-interface is used (see figure 4.19) it then has to be asserted to inform the CPU of the successful transfer. Because the RDY-pin is needed for read-accesses (see below) it has to be considered here, too.

Section 4.2.5.1 already touched the utilisation of the REQ/ACK-protocol to implement write-accesses using the RDY-pin:
First the sending CD (here: the CPU) informs the receiving CD (here: the FPGA) that valid "data" (here: address and data) is available. The receiving side then acknowledges the reception and thereby also permits the sending side to begin with the next cycle. In the case of the VLIO-protocol the $\overline{\text{PWE}}$ signal informs the FPGA that address and data are valid (REQ). The RDY signal then is asserted by the FPGA to inform the CPU of the successful reception (ACK).

Looking back at figure 4.13 it can be seen that a register (clocked by the sending CD) was used to ensure that bus `X` was stable during the protocol cycle. A register serving the identical purpose obviously is integrated in the PXA's memory controller (because address- and data-bus are stable while $\overline{\text{PWE}}$ is asserted). Therefore, it is not compulsory to add such a register inside the FPGA.
However, it is advisable to do so because such a register eases the timing analysis of the FPGA design flow (more details in section 4.3). Hence registers for both address and data were included: `write_data_reg` latches the data bus while `write_reg` registers the address bus and enables-signal generated from the address bus. The latter ones are called `PB_ENABLES` and used to enable one single unit on the PB.

Both the falling and the rising edge on $\overline{\text{PWE}}$ can be used as indicator that address- and data-bus are valid and can be latched by the FPGA safely (see figure 3.2 for reference). However, the analysis of the circuit on the EyeBot M6-PCB in section 3.3.1 showed that both address- and data-bus are routed through (bidirectional) buffers but the transfer qualifiers ($\overline{\text{PWE}}$, $\overline{\text{CS}}$, RDY) are not. These buffers introduce a delay (of up to 5.2 ns) on both buses which is equivalent to shifting address- and data-bus in figure 4.19 to the right (by approx. half a MEMCLK). If the falling edge is used to clock `write_reg` and `write_data_reg` the data only has arrived approx. 4.8 ns before the clock edge. This might be insufficient to conform with the setup requirements of the register because the routing inside the FPGA adds additional delay.
Therefore, it is advisable to use the rising edge of $\overline{\text{PWE}}$ to latch enables-signal, address- and data-bus. This only delays the execution of the write-access inside the FPGA, the achievable data rate will stay the same.

Therefore, the following order of events will take place after address and data have been supplied by the CPU:
The assertion of $\overline{\text{PWE}}$ is recognized by the FPGA and leads to the immediate assertion of RDY because the FPGA "knows" that it is fast enough to handle the

write-access without wait states[17]. This functionality is implemented by `rdymux`. Simultaneously to the latching of enables, address- and data-bus (at the rising edge of $\overline{\text{PWE}}$) the FPGA can be informed that enables, address and data are valid. This is implemented by register `write_en_reg` that latches `PB_BRIDGE_EN` at the same time. `PB_BRIDGE_EN` is generated by the address-decoder in the bus bridge that distinguishes accesses to PB, FIFOs (BB) and the FIFO-control/status-unit (see figure 4.23 for reference).
The output of `write_en_reg` is connected to `X_PB_WR`. A two-stage synchronizer transfers this signal to the FPGA-CD (where it is called `F_PB_WR2`). Based thereon, two flag signals (`F_WR1` and `F_WR2`) consecutively get asserted for one clock cycle each. `F_WR1` is used to latch the enables-signal, the address- and the data-bus into `addr_reg` resp. `wr_data_reg`. Hence `PB_ENABLES`, `PB_A` and `PB_DOUT` are valid after this cycle.
In a final step, `F_WR2` and therefore `PB_WR` get asserted which triggers the unit on the PB to transfer the data on the PB to its internal storage. Additionally, an acknowledge (`X_WR_DONE`) is sent to the CPU-CD. It is used to deassert `X_PB_WR` which brings the unit back to reset state.

In short:

- The CPU first drives chip-select, address- and data-bus
- After that, it asserts $\overline{\text{PWE}}$ for (at least) 4 MEMCLKs
- The FPGA recognizes the write-access and asserts RDY as fast as possible
- The CPU receives RDY and deasserts $\overline{\text{PWE}}$
- This rising edge on $\overline{\text{PWE}}$ clocks enables-, address- and data-bus into registers in the CPU-CD
- A single-bit enable signal then is synchronized to the FPGA-CD
- After the enable signal has been recognized the enables-, address- and data-buses get latched into the FPGA-CD and driven to the PB
- One (FPGA) clock cycle later the FPGA asserts the PB's write signal, the addressed device latches the data on the PB and the unit returns to reset state

Read-accesses are more demanding than write-accesses because the FPGA has to fetch the data requested by the CPU and return it to the CPU-CD before the CPU itself latches the data bus. The underlying procedure however is similar to the write-access described above.
A VLIO-read-access (see figure 3.3) begins with the CPU driving the address bus and $\overline{\text{CS}}$. Two MEMCLKs later $\overline{\text{OE}}$ gets asserted and the RDY-pin is monitored. During this time the FPGA has to evaluate the address and will start fetching the requested data. If RDY is not asserted at that time the CPU will insert (at least) one wait state (see figure 4.20). After the FPGA has finished fetching the data it

[17] It is impossible to assert RDY earlier because the RDY-pin will be driven low during read-accesses and the FPGA is not aware whether a write- or a read-request will take place before either $\overline{\text{PWE}}$ or $\overline{\text{OE}}$ are asserted (RD/$\overline{\text{WR}}$ is not connected).

will assert RDY. Once RDY has been synchronized into the CPU-CD the CPU will latch the data bus and finalize the access by deasserting $\overline{\text{OE}}$.

Again section 4.2.5.1 has already mentioned how to use the RDY-pin to implement the REQ/ACK-protocol for reads:
The two-way-handshake is extended to a three-way one because the receiving CD (the CPU) first requests the sending CD (the FPGA) by asserting $\overline{\text{OE}}$ to send back data. The FPGA then drives the data onto the bus and requests the CPU to fetch the data by asserting RDY (REQ). Finally, the CPU acknowledges the successful transfer by deasserting $\overline{\text{OE}}$ (ACK). The names in brackets again are the equivalent signals of the REQ/ACK-protocol.

For the same reason as already mentioned for write-accesses above it again is advisable to register enables- and address-bus in the CPU-CD. This is handled by `read_reg`. In the case of a read access only the falling edge of $\overline{\text{OE}}$ can be used as indicator that the address bus is valid because the (later) rising edge is delayed until RDY is asserted.

Therefore, the following order of events will take place after the address has been supplied by the CPU:
At the same time the enables- and address-buses have been latched (thus on the falling edge of $\overline{\text{OE}}$) the register `read_en_reg` asserts `X_PB_RD`. This signal then is synchronized to the FPGA-CD. Based thereon, three flag signals (`F_RD1`, `F_RD2` and `F_RD3`) consecutively get asserted for one clock cycle each. `F_RD1` is similar to `F_WR1` and used to enable `addr_reg` which latches enables- and address-buses. Hence `PB_ENABLES` and `PB_A` are valid after this clock cycle.
In the next cycle `F_RD2` and therefore `PB_RD` get asserted. The latter one induces the addressed unit on the PB to drive `PB_DIN` with the data stored at the addressed location. `F_RD2` also enables `rd_data_reg` that latches the data on `PB_DOUT` on the next rising clock edge and drives `X_F2C`.
In a final step, `F_RD3` is asserted which leads to the assertion of RDY by `rd_rdy_reg`. RDY is also used to bring `read_en_reg` back to reset-state. The rising edge of $\overline{\text{OE}}$ furthermore brings `rd_rdy_reg` back to reset-state.

In short:

- The CPU first drives chip-select and address bus
- After that, it asserts $\overline{\text{OE}}$ for (at least) 4 MEMCLKs
- This falling edge on $\overline{\text{OE}}$ clocks enables-signal and address bus into registers of the CPU-CD
- A single-bit enable signal then is synchronized to the FPGA-CD
- After the enable signal has been recognized the enables-signal and the address bus get latched into the FPGA-CD and driven to the PB
- One (FPGA) clock cycle later the FPGA asserts the PB's read signal, latches the data received from the addressed device and drives the data bus towards CPU

- One (FPGA) clock cycle later the FPGA asserts RDY to inform the CPU that valid data is on the bus
- Finally, the CPU latches the data, deasserts $\overline{\text{OE}}$ and the unit returns to reset state

This implementation of the bus interface has multiple advantages compared to the old design analyzed in section 3.3:

Metastability awareness: All (asynchronous) enable signals between CPU-CD and FPGA-CD are transferred using two-stage synchronizers. They increase the mean time between two metastable events reaching the system by magnitudes.

Apart from these asynchronous enable signals three registers with an asynchronous clear-input (`CLR`) are used. Depending on the (unknown) internal structure of the registers these inputs might not be truly asynchronous. If this is the case a timing violation still could occur in one of these registers. Presumably this will not happen in two of them because of the following reasons:
`X_WR_DONE` resets `write_en_reg` after the write access has been processed in the FPGA-CD. As long as the FPGA clock is (much) faster than the maximum frequency of $\overline{\text{PWE}}$ (12.5 MHz) `X_WR_DONE` will be asserted before the next edge takes place on $\overline{\text{PWE}}$ (which clocks `write_en_reg`). Therefore, no timing violation can occur in this register.
`X_RDY` resets `read_en_reg` after the read access has been processed in the FPGA-CD. `PB_BRIDGE_EN` will be stable until RDY has been received by the CPU because the CPU holds the value on the address bus as long as RDY is not asserted. Therefore, no timing violation can occur.

The third register is `rd_rdy_reg` and is reset by `X_NRD` after the CPU has received RDY. Because `rd_rdy_reg` is continuously clocked by the (fast) FPGA clock the output of the register might get metastable from time to time. This however does not impose a problem because this only might happen after a read access has been finished and the RDY-pin is ignored therefore.

Avoidance of race-conditions: All single-bit control signals are synchronized in one single location. Multi-bit signals (e.g. address- and data-buses) are transferred using the REQ/ACK-protocol. This avoids the race conditions presented in sections 4.2.3.2 and 4.2.3.3.

Supporting the synchronous design flow: Looking back at figure 4.24 it can be seen that almost all signals entering resp. leaving the unit (port signals) are latched resp. driven by registers triggered by a clock signal associated with the same port. (The notable exception is `X_F2C` which is generated in the FPGA-CD but is connected to the CPU-port.) This eases the timing analysis using the FPGA's synchronous design flow because (nearly) all timing relations on a particular port then can be described using synchronous timing constraints. More details can be found in section 4.3. Exceptions to this obviously also are the asynchronous control signals "between" the two CDs (`X_PB_RD`, `X_RDY`, `X_NRD`, `X_WR_DONE` and `X_PB_WR`).

4.2.7.3 FIFO control

As already shown in section 4.2.5.2 the VLIO burst mode can be used to generate a clock signal to drive an independent clock FIFO. Looking back at figure 4.21 it can be seen that both address and data are valid on the falling and on the rising edge of $\overline{\text{PWE}}$. But for the same reason already explained in section 4.2.7.2 it is advisable to use the rising edge (to clock the FIFO): Because of the delay introduced by the buffers on address- and data-bus the available time between the arrival of the bus signals and the falling edge might be insufficient.

Additionally, to a clock signal an enable is needed to write data into an independent clock FIFO. Every time this enable is asserted and a rising edge appears on the FIFO's clock input one word will be saved into the FIFO. Because the clock input will be wired to $\overline{\text{PWE}}$ and not all words written to the VLIO shall be stored in the FIFO the FIFO's enable signal `WR_EN` has to be asserted only if an address region assigned to the FIFO is accessed. This is implemented by the address-decoding-unit that generates `ENABLES(1)` (see figure 4.23) and a register that sampled `ENABLES(1)` on the falling edge of $\overline{\text{PWE}}$.

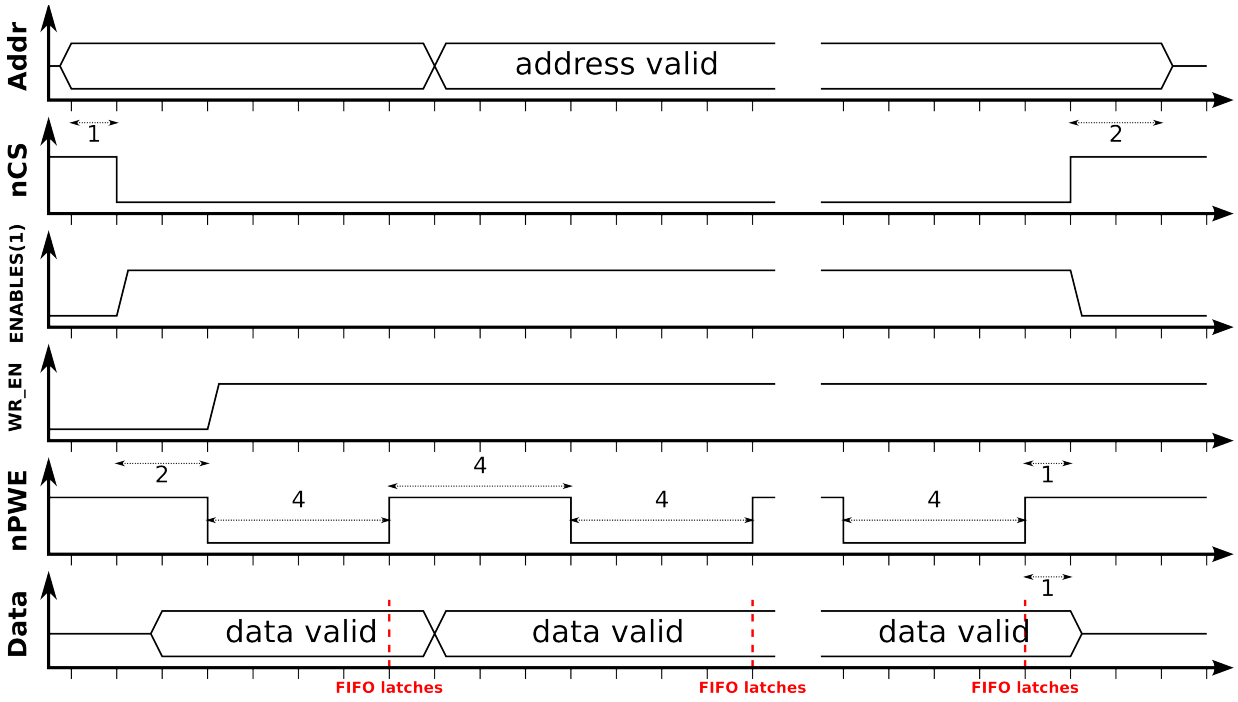

Figure 4.25: VLIO burst write using FIFOs

Figure 4.25 shows the sequence of signals for a VLIO burst write. After the address has been decoded and $\overline{\text{CS}}$ has been asserted the address-decoder will assert `ENABLES(1)`. On the falling edge of $\overline{\text{PWE}}$ the register will sample this signal and drive the FIFO enable signal `WR_EN` to high. On the following rising edges the FIFO will store the data on the data bus. After the burst is finished the CPU will deassert $\overline{\text{CS}}$ and the address-decoder will deassert `ENABLES(1)`.

WR_EN will be deasserted on the falling edge of $\overline{\text{PWE}}$ of a subsequent VLIO write-access to an address region not assigned to the FIFO. This is not depicted in figure 4.25.

Read-accesses are more time-critical because the FIFO has to drive the data bus in a certain time and it is impossible to use the RDY-pin to lengthen the read-access (see section 4.2.5.2 and appendix B). Instead of requesting each single word from the FIFO an independent clock FIFO in "first-word fall-through" (FWFT) (see section 4.2.4) is used: In this mode the FIFO will output valid data as soon as it is available. The surrounding logic then will acknowledge the reception by asserting RD_EN. This signal is as generated as for write-accesses: A register clocked by the falling edge of $\overline{\text{OE}}$ latches ENABLES(1) and drives RD_EN.

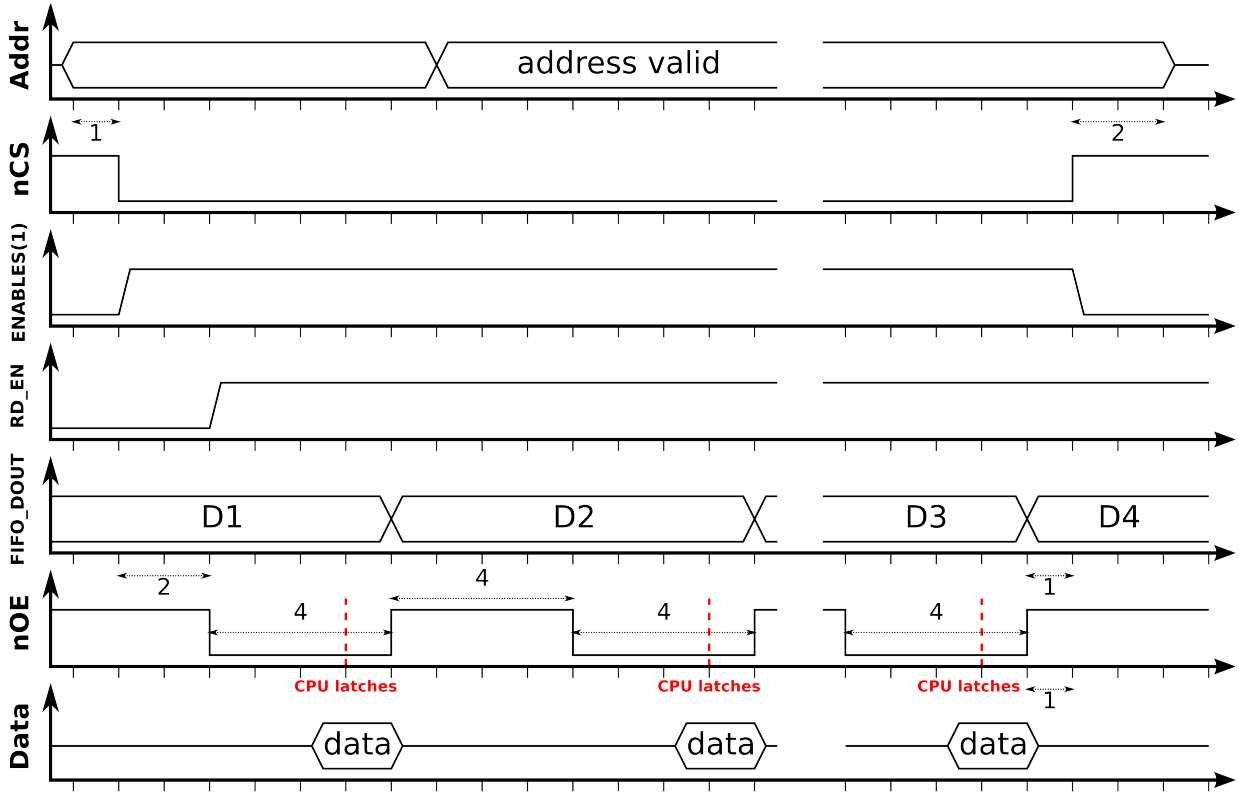

Figure 4.26: VLIO burst read using FIFOs

Figure 4.26 shows the waveform of a VLIO read-access. Apart from the FIFO's data-output (FIFO_DOUT) the timing required by VLIO is depicted (Data, see figure 4.22): The data has to be stable 1.5 MEMCLKs and will be latched 1 MEMCLK before the rising edge of $\overline{\text{OE}}$. This can easily be accomplished because the FIFO outputs the first word of data (D1) as soon as it has been received.

Please note again that this approach only works if it has been ensured that no FIFO-over- resp. -underflow will occur during the burst-access. Therefore, some sort of software-based flow control has to established which was achieved by the FIFO-transactions introduced in section 4.2.5.2. More details can be found in section 4.4.1.6.

4.2.7.4 Evaluation and measurements

While the previous three sections described the implementation of the bus bridge the following evaluates the design, presents transfer rate- and scope-measurements and analyzes the impact of the REQ/ACK-protocol on the achievable speed.

After the bus bridge has been integrated into the design the transfer errors observed previously disappeared. The operativeness of the design was verified by the transfer of gigabytes of data using both small and large FPGA designs. Two tests of the slower REQ/ACK-based interface were undertaken: The correctness of (data) writes and reads was verified using a 16 bit register connected to the PB to which data was written to and read out afterwards. Secondly, the validity of the address bus (`PB_A`) was confirmed by "mirroring" `PB_A` to the data bus (`PB_DIN`) and comparing the received data value with the accessed address.
The test of the FIFO-based interface was performed by connecting the two FIFOs in the FPGA-CD in a way that data written to `cpu2fpga_fifo` could be read straight afterwards from `fpga2cpu_fifo` (loopback). Then the FIFOs were filled and emptied using burst VLIO transfers.

To determine the impact of the REQ/ACK-protocol used for the PB some comparative measurements have been undertaken. Write- and read-accesses to both PB and FIFOs (BB) have been evaluated. Table 4.5 shows the results that were achieved with a simple C program that reads a certain amount of data from the selected interface and measures the throughput. It supports the verification of the received data by comparing it to the (known) expected values. The bus bridge was clocked with 50 MHz during this test.

Method		Interface	
		PB (REQ/ACK-protocol)	BB (FIFO-based)
MMIO	with verify	3.2 MB/s	4.0 MB/s
	without verify	3.0 MB/s	3.0 MB/s
DMA via kernel	with verify	7.2 MB/s	9.6 MB/s
	without verify	7.8 MB/s	13.5 MB/s
DMA-to-userspace	with verify	7.1 MB/s	11.1 MB/s
	without verify	8.7 MB/s	16.2 MB/s

Table 4.5: Achievable transfer rates (read, $f_{FPGA} = 50\,\text{MHz}$)

It can be seen that the verification mostly has a negative impact on the achievable data rate. Oddly enough this does not hold true for MMIO where transfers with verify are faster. The reason therefore presumably is some kind of optimization done by the used compiler. This has not been probed further because such a kind of a read access (reading without evaluating the received data) is not overly close to reality.

In the following a few waveforms of the VLIO transfer qualifiers will be displayed. Channel 1 is connected to $\overline{\text{CS}}$, channel 2 to $\overline{\text{PWE}}$ resp. $\overline{\text{OE}}$ and channel 3 to RDY. The signals were routed to the camera-port to measure the waveforms because it is impossible to

reach the signals on the PCB due to the lack of test-pads. Thus, there might be a (small) phase offset between the different traces depending on the routing inside the FPGA.

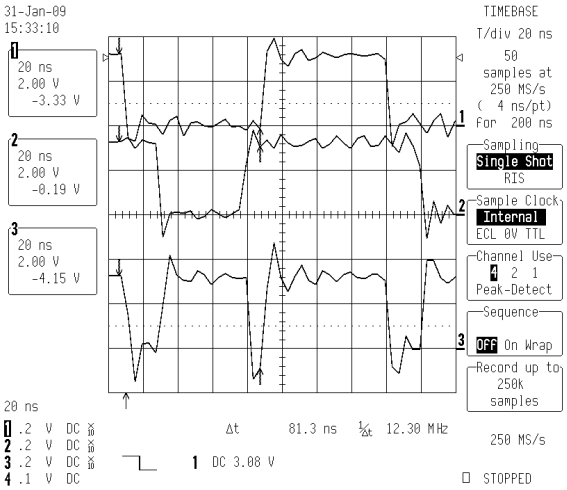

Figure 4.27: PB write using MMIO

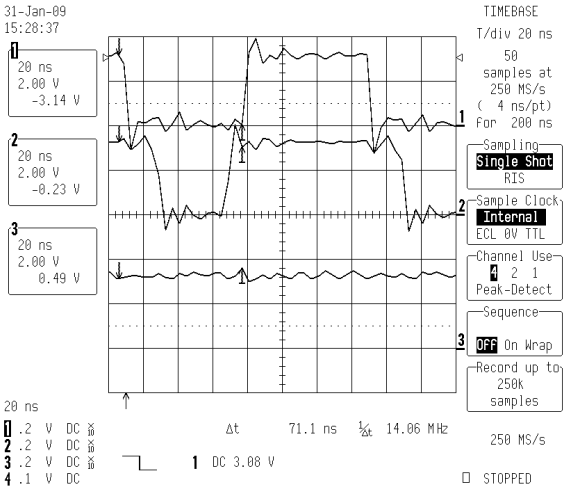

Figure 4.28: BB write using MMIO

Figure 4.27 and figure 4.28 show VLIO writes to PB and BB using MMIO. It can be seen that the RDY-pin is unused during accesses to the BB and the waveform therefore looks similar to figure 3.2 ($\overline{\text{CS}}$ is asserted for 7 MEMCLKs). If the FPGA detects a write access to the PB it asserts RDY immediately and only one wait state is inserted by the CPU ($\overline{\text{CS}}$ asserted for 8 MEMCLKs) and the waveform is similar to figure 4.19.

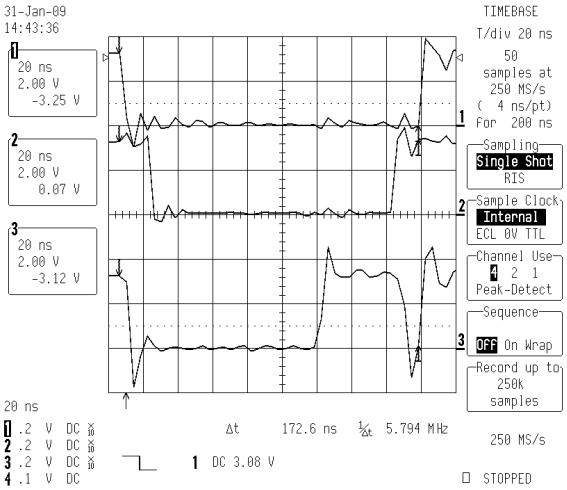

Figure 4.29: PB read using MMIO

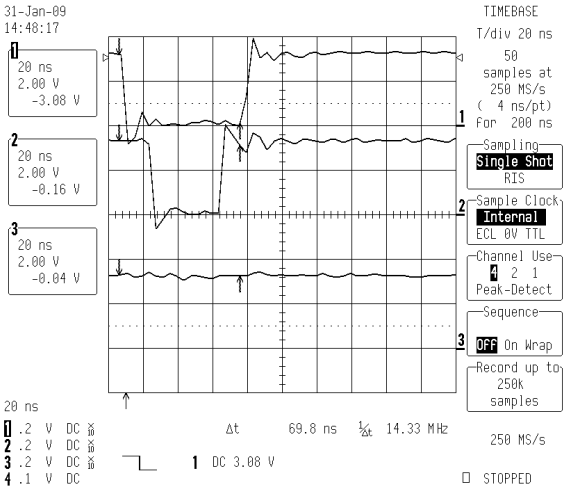

Figure 4.30: BB read using MMIO

Figure 4.29 and figure 4.30 show VLIO reads from PB and BB using MMIO. Again the RDY-pin is unused during accesses to the BB and the waveform therefore looks similar to figure 3.3. If the PB is accessed the RDY-pin is used to lengthen the VLIO bus cycle (see section 4.2.7.2).

Increasing the clock frequency from 50 MHz to 100 MHz should cause higher throughput on the PB because the FPGA will assert RDY earlier. The transfer rates on the FIFO-based interface should be independent on the FPGA's clock frequency (as long as it is higher than the VLIO-frequency). The measurements depicted in table 4.6 confirm this (apart from small variations for kernel-based DMA). Please note that increasing the clock-frequency simultaneously also shortens the clock period and therewith the settling time granted to the synchronizers in the peripheral bus bridge (see figure 4.24 and section 4.2.2.3 for reference), too. Therefore, the MTBF of the FPGA design running at 100 MHz is lower than when running at 50 MHz. This has to be taken into consideration when changing the clock frequency.

Method		Interface	
		PB (REQ/ACK-protocol)	BB (FIFO-based)
MMIO	with verify	3.3 MB/s	4.0 MB/s
	without verify	3.0 MB/s	3.0 MB/s
DMA via kernel	with verify	8.5 MB/s	9.7 MB/s
	without verify	9.4 MB/s	13.6 MB/s
DMA-to-userspace	with verify	8.4 MB/s	11.1 MB/s
	without verify	10.7 MB/s	16.2 MB/s

Table 4.6: Achievable transfer rates (read, $f_{FPGA} = 100\,\text{MHz}$)

It can be seen that read-accesses are significantly slower on the PB than on the FIFO-based interface. Because presumably mostly large, continuous image data will be transferred from the FPGA to the CPU the FIFO-based interface can easily be used to circumvent the negative impact of the REQ/ACK-protocol.

Looking back at section 4.2.7.2 it can be found that write-accesses to the PB can be handled by the FPGA without the insertion of wait states (using the RDY-pin). But because read-accesses to the PB have to be slowed down and the FPGA does not know whether a write- or a read-access will take place the RDY-pin has to be deasserted as soon as $\overline{\text{CS}}$ gets asserted. Therefore, not only read- but also write-accesses to the PB will be slower than the respective access to the BB. The measurements in table 4.7 (for both 50 MHz and 100 MHz) confirm this assumption.

Method	f_{FPGA}	Interface	
		PB (REQ/ACK-protocol)	BB (FIFO-based)
MMIO	50 MHz	9.0 MB/s	9.7 MB/s
	100 MHz	9.1 MB/s	
DMA via kernel	50 MHz	14.3 MB/s	15.6 MB/s
	100 MHz	14.0 MB/s	
DMA-to-userspace	50 MHz	15.2 MB/s	16.6 MB/s
	100 MHz	15.2 MB/s	

Table 4.7: Achievable transfer rates (write)

The obtainable data rate on the FIFO-based interface was exactly the same for both frequencies (which was expected because the RDY-pin is not used here). However, it can be seen that the achievable write-speed on the PB is (apart from minor variations) independent of the FPGA clock frequency. The reason therefore is that (during writes) the RDY-pin is asserted combinatorially by `rdymux` (`bus_bridge_pb`) in the CPU-CD. The delay between the assertion of $\overline{\text{PWE}}$ (by the CPU) and the assertion of RDY (by the FPGA) is therefore only depending on logic- and interconnect-delay and not on the FPGA clock frequency.

It should be pointed out that these measurements were done without any user programs running on the PXA-CPU. If e.g. m6main (the main user interface mentioned in section 2.3) is running the transfer rate drops down to 50%. The reason for this behaviour is not understood yet, therefore it is advisable to stop m6main before running applications that need the full speed of both CPU and VLIO-interface.

Additional measurements can be found in appendix C.

4.2.7.5 Summary

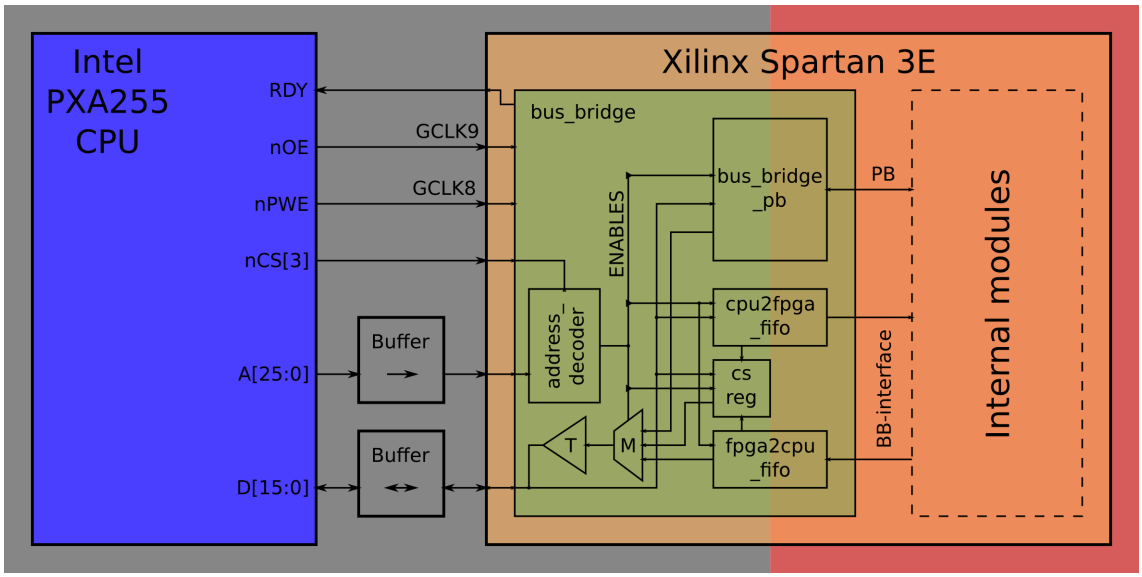

Figure 4.31: Interfacing CPU and FPGA with the bus bridge

Figure 4.31 shows both CPU and FPGA and their respective CDs. The bus bridge implements the CDC using both the REQ/ACK-protocol- and the FIFO-based approach. The former supports arbitrary accesses to devices connected to the PB while the latter facilitates fast transfers at the cost of setup overhead.

The introduction of the bus bridge came with an extensive rewrite of the toplevel-module to hook up the new interface and to instantiate the new bus architecture. For the sake of

clarity several VHDL data types have been introduced to distinguish the various address-parts from each other. Several auxiliary entities have been designed and written from scratch, e.g. an address-decoder, multiplexer-units and a scalable register-file.

Simulation was used to ease the development and the verification of the design. To facilitate not only behavioural but also detailed timing simulation (post-route) a cycle-accurate model of the CPU's VLIO-interface has been developed and integrated into the testbench. It is based on the timing parameters extracted from the kernel driver and from the values given in the PXA-documentation. The testbench not only models VLIO bursts but also the CPU's behaviour when the RDY-pin it used to throttle the transfer rate. Appendix A lists the used parameters. The delay introduced by the buffers on address- and data-bus is modelled, too.
Please note that a simulation reflecting the actual behaviour of the hardware is unachievable because synchronizers are used in the design to facilitate the CDC (see section 4.2.3.4 for reference) and the asynchronous clocks of CPU and FPGA are hard (if not impossible) to model.

4.2.8 Overview of the resulting FPGA system

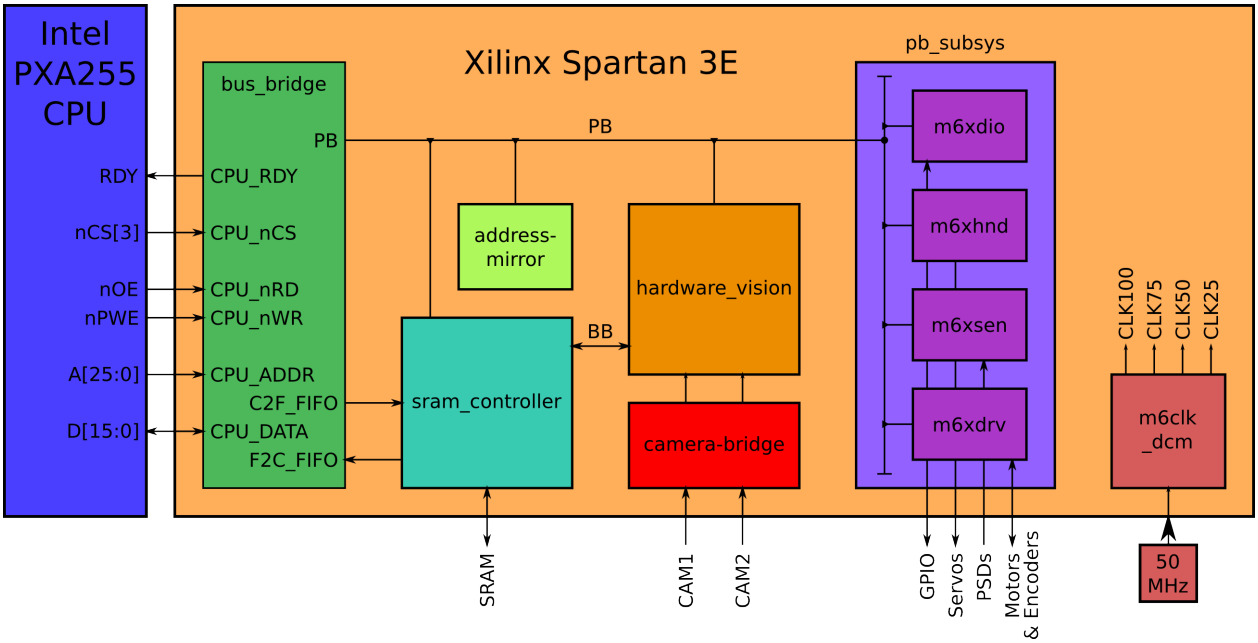

Figure 4.32: Complete system with CPU, FPGA and internal modules

Figure 4.32 shows the complete system consisting of CPU, FPGA and the internal modules. The structure of the SRAM-controller will be elaborated on in section 4.4. The image processing modules in the `hardware_vision`-entity were developed concurrently by a fellow student and are presented in [29].

The implemented design not only handles the asynchronous CPU-interface but also constitutes a clean foundation for further development. The bus bridge takes care of proper synchronization of all signals coming from the CPU and instantiates a fully synchronous internal bus system. Therefore, designers using PB or BB may safely ignore the asynchronous nature of the CPU-interface and rely on the toolchain to detect timing problems. Two distinct approaches to interface the CPU have been deployed, each with different (dis-)advantages. The peripheral bus allows arbitrary access to connected devices and the FIFO-based interface facilitates fast, continuous data transfers from and to the CPU.
Various auxiliary modules were developed and allow subsequent projects to focus on the implementation of new features.

Appendix D lists all entities and data types used in the VHDL design. Appendix E shows the address map of the resulting system. These values should not be used directly, instead thereof the macros in `fpga_busbridge.h` should be used.

To preserve compatibility with the old design (and the existing library) nearly all modules of the old design have been ported to the new one. The only exception is the combined camera- and SRAM-controller that did not satisfy the requirements of the image processing modules.

4.3 Meeting deadlines: Applying timing constraints

Section 3.4 already pointed out that the lack of timing constraints can cause awkward problems during the development. This section starts with an introduction to the timing constraints that are supported by the Xilinx toolchain. After that, these constraints are applied to the new design.

4.3.1 Supported constraint types

The constraints in the following sections will be introduced by means of an example circuit with two clock nets, three inputs and two outputs.

4.3.1.1 Grouping constraints

Most constraints can operate on not only a single element (e.g. a register) but also on groups thereof. Although possible it is not advisable to constrain a design without using groups because they not only improve the readability of the constraints but also shorten the runtime of the toolchain.
Groups can include registers, nets, I/O-pads and multipliers. It is also supported to merge several groups into another and to exclude elements of one group from another.
A common usage scenario for groups is to add all signals of a bus into one group. Then an OFFSET constraint can be used to specify the I/O timing requirements of that bus.

Using group-based constraints for buses allows the toolchain to calculate and output the skew between the bus signals.

Further information on groupings can be found on page 19 of [17]. The Xilinx toolchain already includes predefined groups, they are listed in table 2-1 on the same page.

4.3.1.2 PERIOD

The PERIOD constraint is used to inform the toolchain about period, duty-cycle and jitter expected on a specific clock net that is fed into the FPGA. The toolchain then identifies all synchronous elements (registers, Block-RAMs, multipliers, etc.) inside the FPGA that are clocked by the specified clock net. Finally, it ensures that the path delay (between two such elements) caused by logic and interconnect is shorter than the period of the clock net after clock jitter, clock skew and register setup- and hold-requirements have been taken into account. Additionally, the PERIOD constraint specifies which edge of the clock will be used by default in conjunction with OFFSET constraints. The paths covered by a PERIOD constraint are marked in figure 4.33.

Special rules apply to clock nets that are routed through a DCM. A DCM not only may modify a clock signal in various ways (e.g. phase-shifting, duty-cycle-correction) but also is capable of deriving new (but obviously related) clocks by multiplying or dividing the input clock frequency. If a DCM is fed by a clock signal that is tagged with a PERIOD constraint the toolchain will propagate modified PERIOD constraints to the outputs of the DCM. Further information can be found on page 225 of [17].

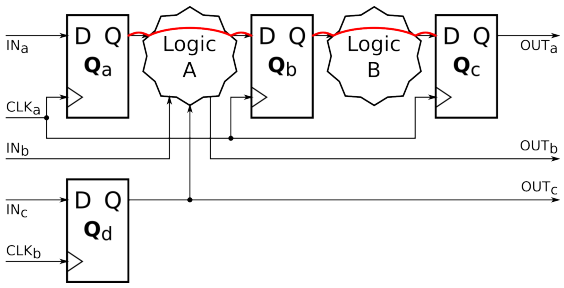

Figure 4.33: Paths covered by a PERIOD constraint

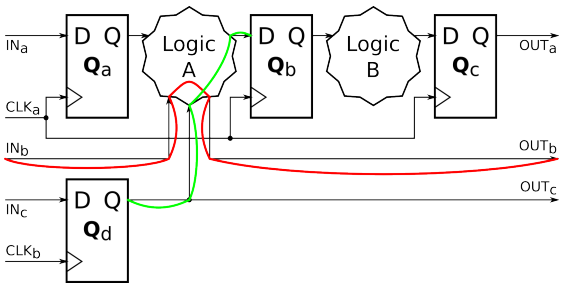

Figure 4.34: Paths (usually) described by a FROM-TO constraint

4.3.1.3 FROM-TO

The FROM-TO constraint is a very powerful tool to limit the path delay between any two groups. It can be used to constrain both paths related and unrelated to a clock net. In the following two typical use cases will be presented.

Pure combinatorial paths can be covered by creating a FROM-TO constraint starting at a group containing one or more input-pads and ending at a group of one or more

output-pads. The red path in figure 4.34 illustrates this case: A pure combinatorial logic block (Logic A) creates a path from signal originating at an input-pin (IN_b) to an output-pin (OUT_b). If the external device that drives IN_b samples OUT_b a certain time T_{delay} after it has changed IN_b a FROM-TO constraint has to be used to limit the path delay to or below T_{delay}. Otherwise a timing violation could occur in the external device.

Paths between two clock domains can be covered by a FROM-TO constraint starting at a register in the sending CD and terminating at a register in the receiving CD. This case is illustrated by the green path in figure 4.34: Register Q_d (clocked by CLK_b) drives a signal ending in Q_b (clocked by CLK_a). Such a path is not captured by a PERIOD constraint because the two registers are clocked by different clock nets and therefore has to be constrained manually.

Further information on the FROM-TO constraint (and related constraints) be found on page 131 of [17].

4.3.1.4 OFFSET IN

The OFFSET IN constraint is used to relate data input-pins to an external clock signal. It specifies the time difference $T_{offset,in}$ between the data at the FPGA's input-pin becoming valid and the arrival of the active clock edge at the FPGA. Apart from $T_{offset,in}$ the name of the associated clock input has to be indicated. Additionally, the time interval T_{valid} in which the data will be valid and the active clock edge (rising or falling) can be given. If the constraint does not specify the active edge the specification in the PERIOD constraint associated with the clock net will be used instead. This information is then utilized to ensure that setup- and hold-requirements of all internal registers (driven by that clock signal and) sampling that input-pin (or a signal depending thereon) are met.

In other words: The toolchain will ensure that the delay on the path from input-pin to the register will be smaller than the time specified in the OFFSET IN constraint after data and clock delay, clock jitter, clock skew and register setup time have been taken into account. Therefore, the register's data input will be stable before the active clock edge arrives. The register's hold time requirement will only be verified if T_{valid} has been specified in the constraint.

The OFFSET IN constraint can be specified in three different ways: Global, group-specific or net-specific. If used globally all inputs-pins of the FPGA will be constrained relative to the specified clock net. If only specified input-pins shall be constrained the group- or even the net-specific approach can be chosen.

Figure 4.35 shows some exemplary paths that are covered by an OFFSET IN constraint: The red path shows an input signal IN_a that is directly fed into a register Q_a that is driven by CLK_a. The same scenario appears on the blue path (IN_c is sampled by Q_d that itself is clocked by CLK_b). The green path marks another type of paths that are

covered by an OFFSET IN constraint: An input signal IN_b traverses through Logic A and finally is registered by Q_b (clocked by CLK_a).

OFFSET IN constraints can be used for both single and double data rate interfaces. Further information can be found on page 206 of [17].

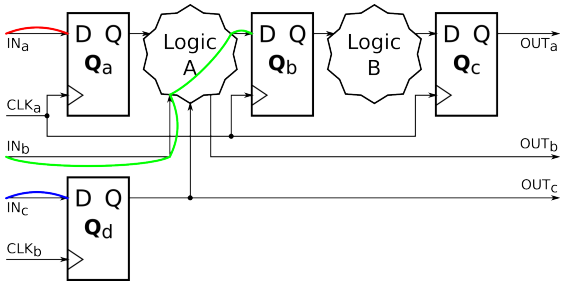

Figure 4.35: Paths described by an OFFSET-IN constraint

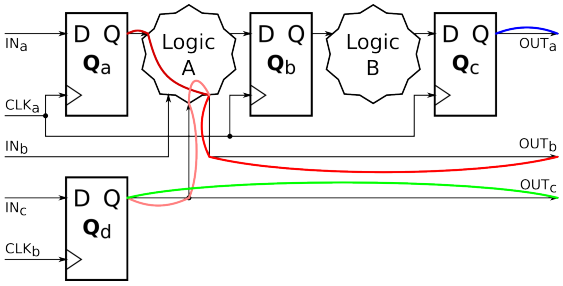

Figure 4.36: Paths described by an OFFSET-OUT constraint

4.3.1.5 OFFSET OUT

The OFFSET OUT constraint is similar to the OFFSET IN constraint but is used to relate data output-pins to an external clock signal. It specifies the time difference $T_{offset,out}$ between the active clock edge at the FPGA's (clock) input and the data becoming valid at the output-pin of the FPGA. Apart from $T_{offset,out}$ the name of the associated clock net has to be indicated and the active clock edge used as reference can be given. If the constraint does not specify the active edge the specification in the PERIOD constraint associated with the clock net will be used instead. An OFFSET OUT constraint is therefore needed to ensure that setup- and hold-requirements of an external synchronous device are met.

In other words: The toolchain will ensure that the delay on the path between the last register and the FPGA's output-pin will be smaller then $T_{offset,out}$ after register clock-to-Q-time, clock delay, clock jitter, clock skew and data routing delay have been taken into account. Therefore, the input of the external device will be stable before the active clock edge arrives there.

As the OFFSET IN constraint OFFSET OUT can be specified in three different ways: Global, group-specific or net-specific. If used globally all output-pins of the FPGA will be constrained relative to the specified clock net. If only specified output-pins shall be constrained the group- or even the net-specific approach can be chosen.

Further information can be found on page 212 of [17].

4.3.1.6 TIG

The TIG constraint can be used to exclude specific paths from the timing-analysis and -optimization. The delay of the ignored paths nevertheless will be calculated and output

by the toolchain. This constraint is handy to "clean up" the timing report generated by the toolchain and allows the developer to focus on more important paths.

4.3.2 Required constraints for the new FPGA design

In the following the constraints needed for the new FPGA design will be presented. Because the asynchronous VLIO bus is terminated in the bus bridge the process of introducing timing constraints was simplified. The reason therefore is that the bus bridge is the only entity that has to be checked for paths that have to be constrained relative to the CPU's VLIO interface. After the groupings have been listed the required constraints of each type will be explained.

4.3.2.1 Grouping constraints

In the following all defined groups will be presented. The following sections then explain the timing constraints that are based on the groups.

Group name	Pin
CLKGRP_M6CLK	M6CLK
CLKGRP_CPU_NWR	CPU_NWR
CLKGRP_CPU_NRD	CPU_NRD
CLKGRP_CAM1_PCLK	CAM1_PCLK
CLKGRP_CAM2_PCLK	CAM2_PCLK

Table 4.8: Clock input groupings via TNM_NET

Table 4.8 shows all net groupings that were created using TNM_NET. These groups will be used to assign a PERIOD constraint to each single clock input. Therefore, every group only contains one net.

The net groupings shown in table 4.9 were created using TNM. These groups are used to identify the nets that have to be constrained using OFFSET IN resp. OFFSET OUT (for synchronous constraints) and using FROM-TO (for combinatorial constraints).
The difference between TNM_NET and TNM is related to the automatic propagation performed by the toolchain. Please see pages 314 and 322 of [17] for reference.

Table 4.10 shows all groupings created to constrain the paths inside the bus bridge that are related to the CDC. These groups are needed for three reasons:

Limit delay on control- and data-paths: To ensure proper functioning of the PB bus bridge the data paths (X_PB_ENABLES, X_PB_A and X_C2F) and the control paths (X_PB_WR and X_PB_RD) have to be of the (approximate) same length. Otherwise it could happen that (if, for example, X_PB_A is much longer than X_PB_RD) addr_reg latches X_PB_A before the new address has arrived.

Group name	Pins
GRP_CPU_NCS	CPU_NCS
GRP_CPU_NWR	CPU_NWR
GRP_CPU_NRD	CPU_NRD
GRP_CPU_RDY	CPU_RDY
GRP_CPU_A	CPU_A<20:1>
GRP_CPU_D	CPU_D<15:0>
GRP_SRAM_A	SRAM_A<19:0>
GRP_SRAM_D	SRAM_D<17:0>
GRP_CAM1_D	CAM1_D<7:0>
GRP_CAM1_CTRL	CAM1_VSYN, CAM1_HREF
GRP_CAM2_D	CAM2_D<7:0>
GRP_CAM2_CTRL	CAM2_VSYN, CAM2_HREF

Table 4.9: I/O pin groupings via TNM

Group name	Path
GRP_BBPB_write_en_reg	write_en_reg → X_PB_WR → synchronizer
GRP_BBPB_write_reg	write_reg → C_PB_WR_ENABLES → mux → X_PB_ENABLES → addr_reg write_reg → C_PB_WR_A → mux → X_PB_A → addr_reg
GRP_BBPB_write_data_reg	write_data_reg → X_C2F → wr_data_reg
GRP_BBPB_read_en_reg	read_en_reg → X_PB_RD → synchronizer
GRP_BBPB_read_reg	read_reg → C_PB_RD_ENABLES → mux → X_PB_ENABLES → addr_reg read_reg → C_PB_RD_A → mux → X_PB_A → addr_reg
GRP_BBPB_synchronizer	F_PB_WR1 and F_PB_RD1
GRP_BB_C2F_FIFO_SEND	Registers inside cpu2fpga_fifo transmitting write- and read-pointers
GRP_BB_C2F_FIFO_RECV	Registers inside cpu2fpga_fifo receiving write- and read-pointers
GRP_BB_F2C_FIFO_SEND	Registers inside fpga2cpu_fifo transmitting write- and read-pointers
GRP_BB_F2C_FIFO_RECV	Registers inside fpga2cpu_fifo receiving write- and read-pointers
GRP_BB_CSREG	all elements in bus_bridge/csreg

Table 4.10: CDC path groupings

Constrain paths inside the independent clock FIFO: Chapter 5, page 91, of [20] suggests that paths inside an independent clock FIFO should be constrained to avoid warnings of the toolchain.

Maximize slack between the registers in the synchronizers: As already explained in section 4.2.3.1 the registers inside a synchronizer have to be placed as close as possible to each other. This maximizes the slack and therefore the time that can be used by the first register to resolve a (possible) metastable state. This finally leads to an increased MTBF of the synchronizer.

To simplify the timing report various paths starting resp. ending at I/O-pins connected to the m6-modules have been ignored. The groups shown in table 4.11 were used therefor.

Group name	Pins
GRP_DIO	DIO<11:9>
GRP_ENC	ENC_A<4:1>, ENC_B<4:1>
GRP_M	MDIR<4:1>, MPWM<4:1>
GRP_SERVOS	SERVO<14:1>

Table 4.11: M6 I/O pin groupings via TNM

Please note: The groupings shown in table 4.8, table 4.9 and table 4.11 are attached to specific nets that are identified by their toplevel names (which are unlikely to be changed). The path based groupings shown in table 4.10 however specify nets and instances based on their position inside the hierarchy. If the bus bridge (and therefore its submodules) is moved all constraints have to be updated. The same applies to net- and instance-names inside the bus bridge. Therefore, it is advisable to avoid any of these changes to ensure that the constraints can be applied successfully.

4.3.2.2 PERIOD

Looking back at figure 4.31 and figure 4.32 it can be seen that five clock nets enter the FPGA. The 50 MHz oscillator generating M6CLK is connected to GCLK11, $\overline{\text{PWE}}$ and $\overline{\text{OE}}$ are wired to GCLK8 resp. GCLK9. Both cameras generate a clock each, too. Five PERIOD constraints were used to specify the clock period on the respective input pin. They are shown in table 4.12.

Name of timespec	Name of group	Frequency
TS_M6CLK	CLKGRP_M6CLK	50 MHz
TS_CPU_NWR	CLKGRP_CPU_NWR	12.5 MHz
TS_CPU_NRD	CLKGRP_CPU_NRD	12.5 MHz
TS_CAM1_PCLK	CLKGRP_CAM1_PCLK	24 MHz
TS_CAM2_PCLK	CLKGRP_CAM2_PCLK	24 MHz

Table 4.12: Deployed PERIOD constraints

Looking back at figure 4.21 and figure 4.22 it can be seen that the minimum period of $\overline{\text{PWE}}$ resp. $\overline{\text{OE}}$ while a DMA transfer is 8 MEMCLKs (80 ns). This is equivalent to a clock frequency of 12.5 MHz. Because DMA transfers are faster than MMIO transfers the latter are covered by these constraints as well.

The data sheet of the camera's CMOS sensor specifies a maximum clock frequency of 24 MHz, but the minimum period is given as 56 ns (equivalent to 17.86 MHz). As a precaution the higher value was used for constraining.

4.3.2.3 FROM-TO

FROM-TO constraints are used to specify both pure combinatorial paths and paths between two CDs. They are shown in table 4.13. t_B is the delay introduced by the buffers on address- and data-bus (see figure 3.1 for reference).

Name of timespec	FROM group	TO group	delay
Combinatorial paths (prefix TS_COMB_) only used for read			
NCS2DATA	CPU_NCS	CPU_D	45 ns - t_B
ADDR2DATA	CPU_A	CPU_D	55 ns - $2t_B$
NRD2DATA	CPU_NRD	CPU_D	25 ns - t_B
Combinatorial paths (prefix TS_COMB_) used for write and read			
NCS2RDY	CPU_NCS	CPU_RDY	20 ns
ADDR2RDY	CPU_A	CPU_RDY	30 ns - t_B
TS_TIG_NWR2RDY	CPU_NWR	CPU_RDY	TIG
Clock domain crossing paths in PB bus bridge (prefix TS_CDC_BBPB_)			
write_reg_TO_addr_reg	BBPB_write_reg	–	10 ns
read_reg_TO_addr_reg	BBPB_read_reg	–	10 ns
write_en_reg_TO_sync	BBPB_write_en_reg	–	10 ns
read_en_reg_TO_sync_AND_muxes	BBPB_read_en_reg	–	10 ns
write_data_reg_TO_wr_data_reg	BBPB_write_data_reg	–	10 ns
synchronizer	BBPB_synchronizer	–	1.5 ns
Clock domain crossing paths in bus bridge (prefix TS_CDC_BB_)			
F2C_FIFO	BB_F2C_FIFO_SEND	BB_F2C_FIFO_RECV	10 ns
C2F_FIFO	BB_C2F_FIFO_SEND	BB_C2F_FIFO_RECV	10 ns

Table 4.13: Deployed FROM-TO constraints

Two constraints are hard to spot: TS_COMB_NCS2RDY and TS_COMB_ADDR2RDY ensure that the RDY-pin is deasserted before the falling edge of $\overline{\text{PWE}}$ or $\overline{\text{OE}}$. As already explained in section 4.2.5.1 the state of the RDY-pin during this falling edge is deciding whether the CPU will lengthen the access by inserting wait states. These paths have to be constrained because $\overline{\text{CS}}$ and the address bus determine whether an access is routed to PB (wait states needed) or BB (no wait states needed).

Please note that (as depicted in figure 3.1) there are two buffers on the path that is constrained by TS_COMB_ADDR2DATA: The address bus leaves the CPU, traverses a buffer and is fed into the FPGA. During a read access the data bus then is driven by the FPGA, traverses another buffer and is finally fed into the CPU.

The CDC paths have been constrained to 10 ns which is a good trade-off between routing complexity and delay. The synchronizer paths have been constrained to 1.5 ns which enforces a close placement inside the FPGA. Additionally, care was taken to avoid the optimization on that paths because otherwise the toolchain would map the synchronizers into a shift register (SRL16) each. This would decrease the MTBF of the synchronizers be-

cause an SRL16 is implemented as a "charge transfer chain" that has a worse metastability behaviour (probably in terms of a longer metastability window and a longer resolution time) than regular registers. More information can be found in [10].

Please note that all the constraints listed in table 4.13 base on the groupings defined in table 4.10. Thus, these constraints depend on the hierarchy of the design and on the net names inside the bus bridge. The consequences were already explained in section 4.3.2.1.

4.3.2.4 OFFSET IN

OFFSET IN constraints are needed for all external signals that are sampled in the FPGA (synchronous to an external clock). The 50 MHz oscillator only is used as clock signal for the FPGA and not fed into any other device. Therefore, no data arrives at the FPGA synchronously to that clock. The CPU address- and data-buses and $\overline{\text{CS}}$ however arrive synchronously to $\overline{\text{PWE}}$ resp. $\overline{\text{OE}}$. The same holds true for the camera data buses and their associated PCLK signals. Therefore, OFFSET IN constraints are needed.

Table 4.14 lists all OFFSET IN constraints that were needed to constrain the synchronous interfaces. The clocks and the relevant clock edges are shown on the left. For each constraint some registers/FIFOs are given that necessitated the respective constraint.

Clock		Group	Offset	Valid	paths via
$\overline{\text{PWE}}$	↑	GRP_CPU_NCS	60 ns	70 ns	`write_en_reg`, `csreg`
		GRP_CPU_A	70 ns - t_B	80 ns	`write_en_reg`, `write_reg`, `csreg`
		GRP_CPU_D	50 ns - t_B	60 ns	`write_data_reg`, `cpu2fpga_fifo`
	↓	GRP_CPU_NCS	20 ns	70 ns	`cpu2fpga_fifo`-enable
		GRP_CPU_A	30 ns - t_B	80 ns	`cpu2fpga_fifo`-enable
$\overline{\text{OE}}$	↓	GRP_CPU_NCS	20 ns	70 ns	`read_en_reg`, `fpga2cpu_fifo`-enable
		GRP_CPU_A	30 ns - t_B	80 ns	`read_en_reg`, `read_reg`, `fpga2cpu_fifo`-enable
CAM1_PCLK	↑	GRP_CAM1_D GRP_CAM1_CTRL	10 ns	30 ns	independent clock FIFO for camera CDC and image stream generator
CAM2_PCLK	↑	GRP_CAM2_D GRP_CAM2_CTRL	10 ns	30 ns	independent clock FIFO for camera CDC and image stream generator

Table 4.14: Deployed OFFSET IN constraints

Example 1: `write_en_reg` samples an enable signal (`PB_BRIDGE_ENABLE=ENABLES(0)`) on the rising edge of $\overline{\text{PWE}}$. This enable signal depends on both $\overline{\text{CS}}$ and the address bus. `read_en_reg` does the same on the falling edge of $\overline{\text{OE}}$.

Example 2: The enable signal for `cpu2fpga_fifo` is generated by a register clocked on the falling edge of $\overline{\text{PWE}}$ that samples `ENABLES(1)` (which depends on both $\overline{\text{CS}}$ and the address bus).

Example 3: `cpu2fpga_fifo` samples the data bus on the rising edge of $\overline{\text{PWE}}$.

There is no OFFSET IN constraint relating $\overline{\text{OE}}$ and GRP_CPU_D because no register reads the CPU data bus during a write-access (which is signalized by $\overline{\text{OE}}$). A relation between $\overline{\text{OE}}$ and GRP_CPU_D is only needed for read-accesses, thus when data is transferred from FPGA to CPU. Since then the FPGA is driving the data bus an OFFSET OUT constraint is needed here.
The timing parameters of the camera were taken from table 13 of the data sheet (which is not available online).

An OFFSET IN constraint can only be used in relation to an external clock input pin of the FPGA. This eliminates the possibility to constrain the SRAM's address- and data-buses because the SRAM is clocked by an output-pin of the FPGA.

4.3.2.5 OFFSET OUT

OFFSET OUT constraints are needed for all signals leaving the FPGA synchronous to a clock that is fed into the FPGA. This is only the case for read accesses where the CPU drives $\overline{\text{OE}}$ (which is fed into the FPGA) and the FPGA drives the data bus. Table 4.15 lists the constraint.

Clock		Group	Offset	paths via
$\overline{\text{OE}}$	↑	GRP_CPU_D	65 ns - t_B	`fpga2cpu_fifo`

Table 4.15: Deployed OFFSET OUT constraints

The offset value of 65 ns can be understood looking back at figure 4.22. `fpga2cpu_fifo` is used in FWFT-mode, therefore valid data will be output by the FIFO as soon it is available. Therefore, the offset has to be measured from the rising edge of $\overline{\text{OE}}$ to the next data word in a burst: After its rising edge $\overline{\text{OE}}$ will be deasserted for 40 ns. Subsequently it will be asserted again. The CPU then requires the data bus to be stable 25 ns after the assertion of $\overline{\text{OE}}$.
Therefore, the data bus has to be stable 40 + 25 ns = 65 ns after the rising edge of $\overline{\text{OE}}$.

There are two more buses leaving the FPGA, the address and the data bus of the SRAM. If the SRAM would be clocked by a clock signal that is fed into the FPGA (and used to trigger the registers driving these buses) OFFSET OUT constraints could be applied. This is impossible though because the SRAM is clocked by an output-pin of the FPGA.

4.3.2.6 TIG

All nets included in the groups shown in table 4.11 were ignored using TIG constraints. This is permissible because all these nets are in- or outputs that are not related to any clock network. Furthermore, it is feasible to ignore the clocked path from `csreg` to the CPU data bus because reads from `csreg` are done combinatorially. This is covered by TS_TIG_BB_CSREG.
Various other paths inside debugging modules were excluded as well.

4.3.3 Summary

The timing constraints introduced in the previous section enforce valid routing inside the FPGA to facilitate reliable communication between CPU and FPGA. If the toolchain fails to satisfy a constraint it will notify the user[18)].

As expected the introduction of timing constraints changed both placement and routing inside the FPGA to meet the required timing. The timing analyzer included in the toolchain supports cross-probing to the FPGA editor which enables the user to trace the constrained paths on the FPGA.

4.4 Providing storage: Accessing the SRAM

The 18 MBit SRAM on the EyeBot M6 has to be utilized as soon as an entire image has to be stored because the internal Block-RAM is too small for it. One of the image processing blocks presented in [29], the stereo rectification unit, needs arbitrary access to two complete camera frames and therefore necessitated the use of the SRAM.
To facilitate easy access to the external SRAM the author of this thesis planned to design an internal bus system dedicated to SRAM accesses. This bus system was named burst bus (BB) and already kept in mind during the design of the CPU interface and the development of the bus bridge (see section 4.2.6 and section 4.2.7 for reference).
Sadly a problem found in the early stages of implementation could not be solved in a timely manner and detained the author of this thesis from implementing the SRAM controller.

Section 4.4.1 therefore presents the ideas behind the burst bus and how it would interact with the image processing units described in [29]. The subsequent section (4.4.2) examines the encountered problems and proposes a solution.

4.4.1 Design of burst bus and SRAM controller

In the following the devised concepts for burst bus and SRAM controller will be presented. After the requirements have been identified the proposed implementation and its use for image processing purposes will be expound. Based thereon, the achievable performance of the complete image processing system will be estimated.

4.4.1.1 Requirements of the image processing system

The requirements on burst bus and SRAM controller heavily depend on the image processing algorithms that shall be implemented in the FPGA. The following applies to the stereo image processing architecture proposed in [29]. It pursues the generation of a depth

[18)] Please note that – at least in the default configuration – the toolchain will only output a warning if a timing constraint is not met. Please check the console output or the timing report for such messages.

map of the scenery recorded by the cameras. Please refer to this document for a more detailed explanation on the deployed approach.

This image processing architecture needs quasi-simultaneous accesses to the SRAM:

1. Both cameras (left and right) continuously stream frames to the SRAM. Because the image processing units operate on grayscale images only it is advisable to use the cameras in YUV mode (the Y channel contains grayscale information). However, it is necessary to save the UV channel as well because the existing RoBIOS-library supports frame grabbing both in color and in grayscale mode and backward compatibility has been requested.
2. The rectification unit used to compensate camera- and alignment-errors fetches umpteen small blocks from both images and generates two rectified grayscale image streams (one for each camera).
3. These image streams are not only stored in the SRAM but also fed into a stereo Harris detector that extracts features from each image. The resulting two lists of features are then stored in the SRAM.
4. Finally, the CPU fetches these feature lists and the rectified grayscale images, correlates the features of both images (using block matching) and generates a depth map of the scenery seen by the cameras.
5. In addition, the user may choose to transfer the (not rectified) color images to the CPU as well (e.g. for reference or demo purposes).

It can be seen that a multiplicity of units have to write to resp. to read from the SRAM. The timing of the system can not be predicted because it consists of four CDs (camera 1, camera 2, FPGA and CPU) with unknown (phase) relationships. Therefore, the burst bus should support dynamic arbitration to handle all these requests as they arrive.

4.4.1.2 Required data flows

Figure 4.37 shows a block diagram of the proposed system consisting of cameras, rectification unit, Harris corner detector, SRAM controller and the CPU link-up through the bus bridge. The four CDs and all the required data flows are marked as well.

All flows defined in figure 4.37 and their data rates are shown in table 4.16. Inside the FPGA both left and right images are processed and transferred in parallel. On the contrary to that the transmission (of left and right frames resp. feature lists) to the CPU is done in a serial fashion.

The BB provides several transaction types that define the kind of the transferred data. Details on the individual transaction types can be found in section 4.4.1.4.

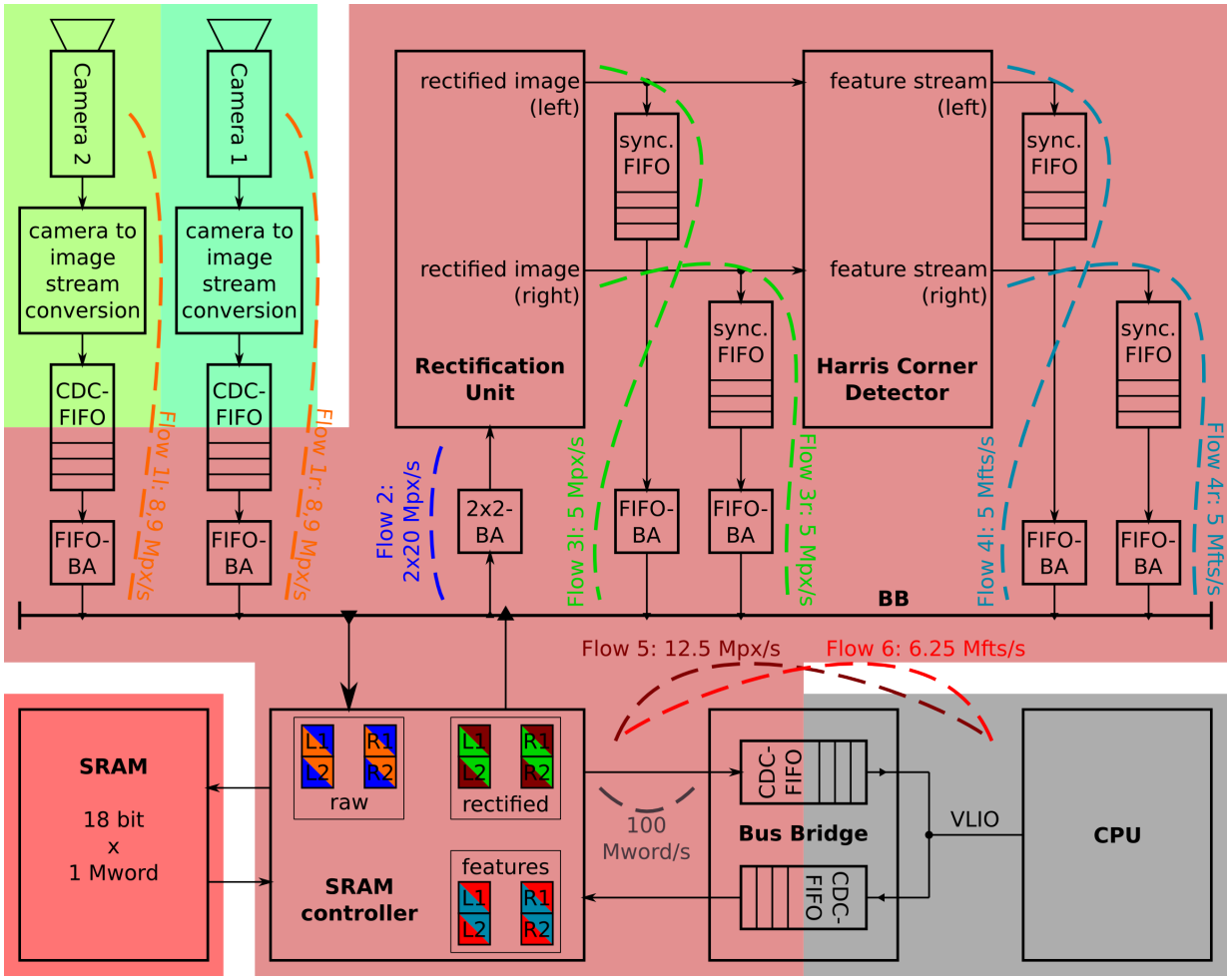

Figure 4.37: Image processing system and data flows

	Source	Data	word size	Rate	Sink
1l	Camera 2	raw image left (color)	16 bit (YUV)	8.9 Mpx/s	SRAM
1r	Camera 1	raw image right (color)	16 bit (YUV)	8.9 Mpx/s	SRAM
2	SRAM	raw image (left and right channel alternating)	16 bit (YUV)	2×20 Mpx/s	rectification unit
3l	rectification unit	rectified image l. (gray)	8 bit (Y)	5 Mpx/s	SRAM
3r		rectified image r. (gray)	8 bit (Y)	5 Mpx/s	SRAM
4l	Harris detector	feature list left	18 bit	5 Mfts/s	SRAM
4r		feature list right	18 bit	5 Mfts/s	SRAM
5	SRAM	rectified image (l. resp. r.)	8 bit (Y)	12.5 Mword/s	CPU
6	SRAM	feature list (l. resp. r.)	2×9 bit	12.5 Mword/s	CPU
7	SRAM	raw image (l. resp. r.)	16 bit (YUV)	12.5 Mword/s	CPU

Table 4.16: Data flows and estimated transfer rates on the BB

The processing can be split into three sub-processes:

Image capturing: The cameras output the YUV data as a repetitive pattern of 8 bit values (sequence Y U Y V) with a byte rate of 17.8 MHz each. One pixel consists of 8 bit Y and 8 bit U resp. V which is converted to a so called image stream with a width of 16 bit and a pixel rate of 8.9 MHz. These two streams are stored in the SRAM (flows `1l` and `1r`).

Image processing: The images are then read by the rectification unit. One pixel of the destination image is interpolated from four source pixels, therefore the input data rate will be four times higher than the output data rate. The theoretical maximum output rate is 20 Mpx/s[19], but because of the arbitration and setup overhead introduced by the BB the actual rate will be lower. This overhead (roughly estimated in section 4.4.1.8) will reduce the output rate to 10 Mpx/s or less. Because two images will be processed in parallel the resulting maximum rate is 5 Mpx/s for each image channel. The input data rate for each channel therefore is 20 Mpx/s (flow `2`). Due to hardware constraints the rectification unit was designed to operate on the Y channel only (U and V are transferred in flow `2` but discarded by the unit). The rectified (grayscale) image generated by the rectification unit is then saved back to the SRAM (flows `3l` and `3r`) and simultaneously fed into the Harris detector. The Harris detector then outputs one feature list for each image channel that is saved in the SRAM (flows `4l` and `4r`).

Transfer to CPU: The CPU finally fetches both rectified grayscale images (two times flow `5`) and the two feature lists (two times flow `6`) from the SRAM. If required flow `7` (not shown in figure 4.37) may be used to fetch the unrectified color images. These transfers are done sequentially using the FIFO-based interface presented in section 4.2.5.2. The maximum transfer rate on the VLIO-interface is 12.5 Mword/s, the FIFOs however can be read resp. written with 100 Mword/s.

4.4.1.3 Required high level control

A high-level flow control for the image processing modules is needed to ensure that both cameras have stored one frame each into the SRAM before the rectification unit starts its calculations. This could be achieved using a double buffering approach: Four frame buffers (for the raw images) are created in the SRAM, two for the left camera (L_1 and L_2) and two for the right camera (R_1 and R_2). After L_1 and R_1 have been filled by the cameras the control redirects their data streams to L_2 resp. R_2 and the rectification unit starts processing the frames in L_1 and R_1. After the processing has been finished the cameras are switched back to L_1 and R_1 and the rectification unit starts processing L_2 and R_2. This ensures that the most recent camera frames are used for rectification. If the exact runtime of the rectification unit is known it is also possible to stop the cameras

[19] Five clock cycles (at 100 MHz) are needed to generate one pixel of the destination image: One to request the four source pixels and four for the actual transmission from SRAM to rectification unit. More information can be found in section 4.3 of [29].

while the rectification unit is busy. The cameras then could be reactivated in such a way that two new frames are available in L_2 resp. R_2 when the rectification has been finished and a new rectification pass could then be started immediately.
The same scheme is needed for the rectified grayscale images generated by the rectification unit and for the feature lists generated by the Harris detector. They will be fetched by the CPU after the processing inside the FPGA has been finished and the CPU has been informed by high-level control thereof.
Therefore, the SRAM has to support the storage of (at least) 8 frames and 4 feature lists in total. The respective storage locations are indicated in figure 4.37 by the small colored squares sketched into the SRAM controller. The colors signalize the flows associated to that particular location.

Alternatively an easier, but slower fully sequential approach could be deployed by stopping the cameras until the full processing has been done and the CPU has fetched both images and feature lists.

4.4.1.4 Proposed transaction types and address generation

The SRAM supports clock frequencies up to 100 MHz for both read and write accesses. It is advisable to use this frequency to exploit the SRAM's full throughput. Because SRAM and BB have to form a synchronous system the frequency of BB and all connected units will be 100 MHz, too. Therefore, 100 Mword/s (more than 200 MB/s) can be transferred on the BB (to resp. from the SRAM).

One drawback of a multi-master bus system (see section 4.4.1.5) is the additional time needed for the arbitration. Even if the bus is idle (and the requesting device therefore could use it immediately) the arbitration procedure has to be performed to ensure that no other units start using the bus at the same time. This additional time results in an overhead that decreases the achievable (average) data rate.
To alleviate the impact of this overhead only large, continuous transfers ("bursts") should be performed on such a bus system. The longer the transfer the smaller is the arbitration overhead as a percentage. These transfers have already been mentioned in section 4.2.6 and will be called (BB-)transactions.
Looking back at figure 4.37 six FIFOs can be identified. They decouple data sources and BB which is required to adapt the data rate of a particular source to the data rate on the BB[20]. They are required because all units that write to the SRAM (cameras, rectification unit and Harris corner detector) are not able to output 100 Mword/s and therefore would not be able to perform a burst transfer that saturates the BB. After the FIFO fill level has reached a certain limit (depending on the particular flow) a transaction on the BB will be initiated and the FIFO will be emptied ("flushed") using the BB's native data rate of 100 Mword/s. This obviously is more efficient that requesting several smaller transactions with lower data rates.

[20] The two FIFOs connecting the cameras to the BB serve an additional purpose: They implement the CDC between the two camera-CDs and the FPGA-CD.

It is advisable to use a word size of 18 bit on the BB because this matches the SRAM's native word size[21]. Therefore, all transfers on the BB may use a word size up to 18 bit. The width of the CPU data bus however is only 16 bit, therefore transfers involving the CPU may only utilize 16 bit or less.
Because the cameras output 352×288 px in YUV using 4:2:2 chroma subsampling one pixel consists of 1 byte brightness (Y) and 1 byte color (U resp. V) information. Therefore, one color pixel (16 bit) can be saved per SRAM word (18 bit). If a frame shall be transferred to the CPU the 16 bit data bus is wide enough.
This is different for the feature lists though: One feature is identified by its x- and y-coordinates and each coordinate has to be represented by 9 bit because both maximum coordinate values (352 resp. 288) are in the range [256; 511] (representable by 9 bit). One feature therefore needs 18 bit of storage. This perfectly matches the word size of the SRAM but poses a problem for the transfer of the feature lists to the CPU because the width of the CPU data bus is only 16 bit. Therefore, the feature coordinates have to be rearranged before they are sent to the CPU. This can be achieved with two distinct transaction types for feature lists: One is used to store the coordinate tuples as 18 bit words into the SRAM. The other one is used to read one coordinate tuple (consisting of two 9 bit coordinates) as two 16 bit values that then can be sent to the 16 bit data bus.

	Name	Description	Length (words)	address-ed by	used by flows
1	2 lines (YUV)	704 px color (2 byte per px)	704	PSID, line (y)	`1l`, `1r`, `7`
2	2 × 2 window (YUV)	window containing 4 px (color)	4	PSID, x and y	`2`
3	2 lines (Y only)	704 px rectified grayscale (1 byte Y and 1 byte unused)	704	PSID, line (y)	`3l`, `3r`, `5`
4a	512 features	512 coordinate tuples (9 bit for the x-coordinate, 9 bit for y)	512	LSID	`4l`, `4r`
4b			1024	LSID	`6`

Table 4.17: Proposed transaction types on BB resp. FIFO-based interface

Table 4.17 lists the transaction types (TT) that are needed for the proposed system.

TT 1 is used in flows `1l` and `1r` to transfer the color images from cameras to SRAM.
The rectification unit then issues one request of TT 2 (flow `2`) for each pixel in each (rectified) destination image. One pixel of each rectified destination image then will be interpolated based on four pixels of the associated source image. Even tough U and V are fetched only the Y channel is rectified by the unit.
The rectified grayscale images are not only saved to the SRAM using TT 3 (flows `3l` and `3r`) but also fed into the Harris corner detector.
Each feature (with a width of 18 bit) output by the Harris corner detector is saved back to the SRAM using TT 4a (flows `4l` and `4r`). As already explained above the coordinates of one feature have to be realigned for the transfer to the CPU (TT 4b, flow `6`).

[21] The SRAM is organized as 1 Mword × 18 bit and thus is addressed by a 20 bit address bus.

The CPU finally reads both rectified grayscale images using TT 3 (flow 5) to perform the block matching (based on the features transferred previously).
In case the original non-rectified color images are needed TT 1 can be reused (flow 7 from SRAM to CPU, not depicted in figure 4.37).

Usual bus systems implement bursts in a way that either the master transfers an address for every single beat or the slave internally generates increasing addresses based on a start address supplied by the master.
Neither of these approaches was chosen for the BB. It seemed more appropriate to define several transaction types (see table 4.17) and to generate the needed addresses (supplied to the SRAM) in the SRAM controller. A transaction type shall therefore not only determine the type of the transferred data but also the initial and subsequent addresses (sent to the SRAM). The reasons therefore will be given in the following.
All the transaction types mentioned above transfer more than one word in a row. Therefore, it is unnecessary to support word-based accesses to resp. from the SRAM on the BB. The SRAM will be split into several storage areas, e.g. into 8 slots for color images and 4 slots for feature lists[22)] (which would perfectly match the requirements shown in section 4.4.1.3).
If e.g. TT 1 resp. TT 3 are used to transmit two lines of a (color resp. grayscale) image the only information needed is the number of the image slot (PSID, picture slot ID) and the line number y that should be operated on. For TT 2 only PSID and x- and y-coordinates are needed. Feature lists (TT 4a and 4b) are addressed by list slot ID (LSID).

This transaction-type-based approach has several advantages in the proposed system:

- The various masters on the BB only have to notify the SRAM controller of the required TT and of the specific data set that should be operated on.
- The SRAM controller partitions the SRAM into suitable memory areas and generates the word addresses needed to access the SRAM.
- Therefore, only one address generation unit is needed in the whole system and the complexity of the units on the BB is reduced.
- This address generation unit can be placed close to the FPGA's I/O-pins to improve the timing.

The generation of the word addresses needed to communicate with the SRAM is a complex task because one image consists of $n_{pixel} = 352 \times 288 = 101376$ px which is no power of two. Therefore, the 20 bit address bus of the SRAM can not simply be splitted. To calculate the word address at least one multiplier is needed which not only occupies FPGA resources but also introduces additional latency.

For TT 1 and TT 3 (addressed by PSID and y) the following has to be done: To align the eight picture slots without any cut-offs a multiplication has to be used to generate the

[22)] The 8 slots for color images resp. feature lists will occupy $8 \cdot 352 \cdot 288 = 811008$ resp. $4 \cdot 512 = 2048$ words. If this partitioning is used still $2^{20} - 8 \cdot 352 \cdot 288 - 4 \cdot 512 = 235520$ words (529920 byte) are free for other purposes.

start address of the PS ($\alpha = \text{PSID} \cdot n_{pixel}$). Another multiplication is needed to calculate the line offset inside the image ($\beta = y \cdot 352$). The sum of both then is the word address $\text{ADDR} = \alpha + \beta = \text{PSID} \cdot n_{pixel} + y \cdot 352$ of the first pixel in the requested line. If this is implemented with one multiplier three clock cycles are needed to generate the address.

For TT 2 (addressed by PSID, x and y) the situation is similar: The start address of the PS is calculated the same way ($\alpha = \text{PSID} \cdot n_{pixel}$). Again another multiplication is needed to calculate the line offset ($\beta = y \cdot 352$). At the same time an interim result can be calculated ($\gamma = \alpha + x$). The sum then is the word address $\text{ADDR} = \gamma + \beta = \text{PSID} \cdot n_{pixel} + x + y \cdot 352$ of the top left pixel (in the requested 2x2-window). If this is implemented with one multiplier three clock cycles are needed to generate the address.

The situation is easier for TT 4a because its length (512) is a power of two. Therefore, bits 11 and 10 of the 20 bit SRAM address can be used to address one of the four list slots. Only one clock cycle is needed to add the LSID to the base address of the list slots.

The impact of this additional latency will be analyzed in section 4.4.1.8.

4.4.1.5 Proposed arbitration scheme

As found in the previous sections the burst bus has to be able to serve multiple requests from different units quasi-simultaneously. Units that initiate transfers on a bus system are called (bus) masters, responding units are called (bus) slaves. The only slave in the proposed system is the SRAM controller that serves the masters' requests by forwarding them to the SRAM. All the image processing units mentioned above are masters, the BB therefore is a multi-master bus system (as already deduced in section 4.2.6).
Multi-master systems generally require some kind of arbitration to determine which master is the next one allowed to initiate a transfer on the bus. The process of arbitration generally takes some time and thus creates an overhead (compared to a single-master system without arbitration).

The deployment of a static predefined arbitration scheme would waste precious data rate because the requests from different units (located in different CDs) will arrive in a non-predictable, varying manner and it would be necessary to assign a certain percentage of the available bus bandwidth to each unit. In case of a unit being idle the assigned bus bandwidth would be unused and thus lost. In addition, a predefined arbitration scheme has to be adapted if new units are connected.
Therefore, a simple, dynamic priority-based arbitration scheme was selected for this system. The advantage is that bus bandwidth dynamically gets assigned to a specific unit only if that unit actually has some data to transfer. On the other hand this scheme is endangered to data loss if the bus is overloaded temporarily and if the units are unable to buffer the data they want to transfer until the bus is able to handle their request.

To reduce the risk of data loss the priorities would be assigned as follow (highest to lowest):

1. Left camera
2. Right camera (cameras can be interchanged)
3. The output module of the rectification unit (writing the two rectified images)
4. The output module of the Harris corner detector (writing the two feature lists)
5. The input module of the rectification unit (reading the two raw camera frames)
6. The CPU interface used to fetch feature lists, rectified grayscale images and (optionally) non-rectified color images

The highest priorities will be assigned to the cameras because they continuously output data and can not be stopped after the transmission of a frame has been started[23)].
The next lower priorities will be assigned to the modules that write the two rectified image streams resp. the two feature lists to the SRAM. The module fetching the unprocessed, raw frames into the rectification unit will be assigned an even lower priority. This is needed because the image processing units have been implemented with forward flow control only: It is possible to stall the processing if no more input data is available at the moment, though it is impossible to stop the processing if the downstream units (receiving images resp. feature lists) are not ready to receive. By assigning higher priorities to the modules writing the results (compared to the unit reading the input data) it can be ensured that no overflow will occur in the output modules because the rectification unit (being out of new input data) will stall the processing.

4.4.1.6 Proposed CPU link-up using the FIFO-based interface

The lowest priority on the BB will be assigned to the CPU-to-BB interface. Looking back at section 4.2.5.2 it can be seen that CPU accesses to the SRAM shall be implemented as FIFO-transactions (to establish a software flow control on the VLIO bus). These FIFO-transactions on the VLIO bus can be mapped to requests on the BB:

Write accesses are handled as follows: The CPU first checks whether FIFO1 is empty.
If FIFO1 is empty the CPU supplies the address (to be written to) to the FPGA and writes the actual data into FIFO1. The FIFO-transaction which is defined from the CPU's point of view is finished at this point in time. The data stored in FIFO1 however still has to be transferred to the SRAM. Therefore, the CPU-to-BB interface requests access to the BB and finally writes the data (from FIFO1) to the SRAM (after it has been granted access to the BB).
If FIFO1 is not empty the previous write access has not been processed by the CPU-to-BB interface yet and the CPU has to wait until the FPGA has handled the previous write.

[23)] Of course it is possible to stop the cameras from grabbing frames, but it is impossible to stop the transfer "in the middle" of a frame. The reason therefore is that the cameras stream the image data directly from the CMOS sensor without intermediate buffering.

Read accesses are handled as follows: The CPU firsts transfers the address (to be read from) to the FPGA and requests the CPU-to-BB interface to fill FIFO2 with the desired data. This causes the CPU-to-BB interface to request access to the BB. After access has been granted the interface fetches data from the SRAM and stores it in FIFO2. Finally, the CPU is informed that enough data is available in FIFO2 and the CPU starts reading.

It can be seen that the FIFO-transactions on the VLIO bus can easily be used by the CPU-to-BB interface to support the arbitration on the BB.

4.4.1.7 Proposed partitioning

The storage architecture provided by the BB will be implemented using several modules already depicted in figure 4.37. The SRAM controller couples BB and SRAM and the various bus adapters (BA) link the bus masters to the BB.

The SRAM controller is the only slave on the BB. It takes care of the arbitration and handles requests issued by the masters. Thereto, it converts the ID-based addressing of the requests into word addresses that are then used to communicate with the SRAM.

The FIFO bus adapter (FIFO-BA) fetches data from a FIFO and stores it into a predefined storage slot in the SRAM. It monitors the FIFO's fill level and starts a BB-transaction as soon as a certain FIFO fill level has been reached.
FIFO-BAs are used for both images and feature lists, hence configuration options for both TT and the associated slot IDs have to be provided.

The 2x2-BA is only used by the rectification unit to fetch four pixels from a 2x2-window that are needed for the rectification algorithm. The rectification unit only transmits the x- and y-coordinates of the top left pixel and the PSID of the desired image. The SRAM controller then automatically calculates the (word) addresses of the four pixels, reads the pixels from the SRAM and transmits them to the rectification unit.

The SRAM controller itself consists of three submodules:

A combinatorial arbiter which is connected to every master on the bus. It outputs the number of the master with the highest priority that currently requests bus access.

An address generation unit that decodes the TT and its addressing information (PSID, LSID, x- and y-coordinates). It counts the number of words already transferred and generates the word addresses required for the SRAM.

An FSM that controls the BB. It reads the arbitration result from the arbiter, grants access to the BB, initiates the address calculation and handles the communication with the SRAM.

The BB itself has to be implemented as a large multiplexer because no internal tri-states are available in the FPGA.

4.4.1.8 Performance estimations

The delay introduced by address generation (see section 4.4.1.4) and data fetching influences the maximum pixel rate than can be generated by the rectification unit. Because this rate has a big influence on the achievable frame rate it will be estimated in the following. Based thereon, the achievable frame rate of the whole system will be calculated.

	rectification unit	SRAM controller	SRAM
1	requests transaction (TT 2, PSID, x and y)	arbitration (combinatorial)	idle
2	wait for data	arbitration finished, fetch request (PSID, x and y)	
3		$\alpha = \text{PSID} \cdot n_{pixel}$	
4		$\beta = y \cdot 352$, $\gamma = \alpha + x$	
5		ADDR1 $= \gamma + \beta$	
6		ADDR1 sent to SRAM, calculate ADDR2	output D1
7	D1 arrived	D1 latched, ADDR2 sent, calculate ADDR3	output D2
8	D1 latched D2 arrived	D2 latched, ADDR3 sent, calculate ADDR4	output D3
9	D2 latched D3 arrived	D3 latched, ADDR4 sent	output D4
10	D3 latched D4 arrived	latch D4	idle
11	D4 latched	idle (ready for new transaction)	

Table 4.18: Order of events during a transaction of TT 2

One transaction of TT 2 is needed for every rectified output pixel. Table 4.18 shows the order of events during a transaction of TT 2. It can be seen that a transaction may be started every 10 clock cycles (10 MHz). Because each transaction transfers four pixels the resulting pixel rate is 40 Mpx/s as already shown in table 4.16 (flow `2`). The efficiency of a burst transfer is defined as $(T_{transfer})/(T_{transfer} + T_{setup})$. Transactions of TT 2 therefore have an efficiency of only $(4)/(4 + 6) = 40\%$. Transactions of all other TTs have a much higher efficiency because much more words are transferred in one transaction (see table 4.17 for reference).
Please note that this calculation is based on the assumption that flow `2` is the only flow on the BB. In reality both flows `3l` and `3r` and flows `4l` and `4r` will decelerate the transfer.

In addition, all delays originating in arbitration delays (that will occur due to congestion on the BB) will be neglected because they can not be estimated reasonably. This means that all flows involved in the image processing sub-process (`2`, `3l`, `3r`, `4l` and `4r`) can be treat as independent subsequent transfers that may fully utilize the BB. As determined

above the maximum rate of flow `2` is 40 Mpx/s. The maximum rate of flows `3l` and `3r` resp. flows `4l` and `4r` is close to the BB's maximum rate of 100 Mword/s[24)].

Under these assumptions the upper bound for the achievable frame rate can be calculated:

Image capturing: Each camera streams one image (101376 px) into the SRAM with a rate of 8.9 Mpx/s. Thus, one transfer will take 101376 px/8.9 Mpx/s $\approx$ 11.4 ms. Both cameras capture at the same time, therefore the whole process of image capturing is finished after this time.

Image processing: The rectification unit fetches four pixels for every pixel in one output image and therefore reads 8 full images to generate the two rectified output images. Assuming the whole transfer capacity of the BB will be used for flow `2` a transfer rate of 40 Mpx/s can be achieved. Therefore, the rectification of both images will take $8 \cdot 101376$ px/40 Mpx/s $\approx$ 20.3 ms.
The transfer of the rectified images will take $2 \cdot 101376$ px/100 Mpx/s $\approx$ 2.0 ms (flows `3l` and `3r`). The two feature lists are much smaller than the images and thus can be neglected (flows `4l` and `4r`).

Transfer to CPU: The maximum transfer rate on the CPU interface is 12.5 Mword/s, therefore the two rectified images will be transferred in $2 \cdot 101376$ px/12.5 Mpx/s $\approx$ 16.2 ms. The transfer of the two feature lists can be neglected here as well (flow `6`).

Consequently the full processing (capturing, rectification, feature extraction and transfer to the CPU) is finished after approx. 50 ms. This results in an achievable frame rate of 20 frames/s. It can be seen that the resulting overall duration is dominated by transfer times. This highlights the negative impact of using one (external) single-port SRAM instead of multiple independent (internal) dual-port Block-RAMs (that would have been available in a more recent FPGA).

4.4.1.9 Summary

The image processing system presented in [29] required the storage of (at least) two camera frames to perform the generation of a depth map using the two cameras of the EyeBot M6. Therefore, the image processing system's storage requirements were analyzed and the expected flow of data to and from the SRAM was derived thereof. In addition, the high level control required to manage storage locations and full system (including cameras and CPU) was addressed.
The proposed storage architecture consisting of burst bus (BB) and SRAM controller supports quasi-simultaneous SRAM accesses. Competing concurrent transfer requests are handled by the controller using a simple priority-based arbitration scheme. Mostly long transfers will be performed on the BB to mitigate the negative impact on the achievable transfer rate caused by arbitration and setup overhead. The address generation is handled

[24)] Looking back at table 4.18 it can be seen that 6 clocks go by before data is transferred. This time is needed to setup the transfer and to generate the required word address. The efficiency is therefore equal to $\frac{704}{704+6} = 99.15\%$ and the overhead can be neglected.

by the SRAM controller to save resources and improve the timing inside the FPGA. In addition, several bus adapters (simplifying the connection to the image processing units) and the link-up to the FIFO-based CPU interface has been introduced.

The achievable processing speed of the system was estimated based on the speed of the image processing units and on the expected overhead of the storage architecture. Under the (unrealistic) assumption of no bus congestions (mutually caused by the image processing modules) a maximum average processing rate of 20 frames/s can be achieved.

4.4.2 Discovered problems

As already mentioned in sections 4.3.2.4 and 4.3.2.5 the SRAM can not be constrained in a straightforward way. The reason therefore was that the toolchain does not support OFFSET constraints relative to an output of the FPGA. This poses a problem because the clock input of the SRAM is connected to an output pin of the FPGA.

In addition, a serious problem related to the clocking of the SRAM has been found. This problem will be examined in section 4.4.2.1. A solution (that also solves the problems related to timing constraints) will be proposed in section 4.4.2.3.

4.4.2.1 Clock skew between FPGA and SRAM

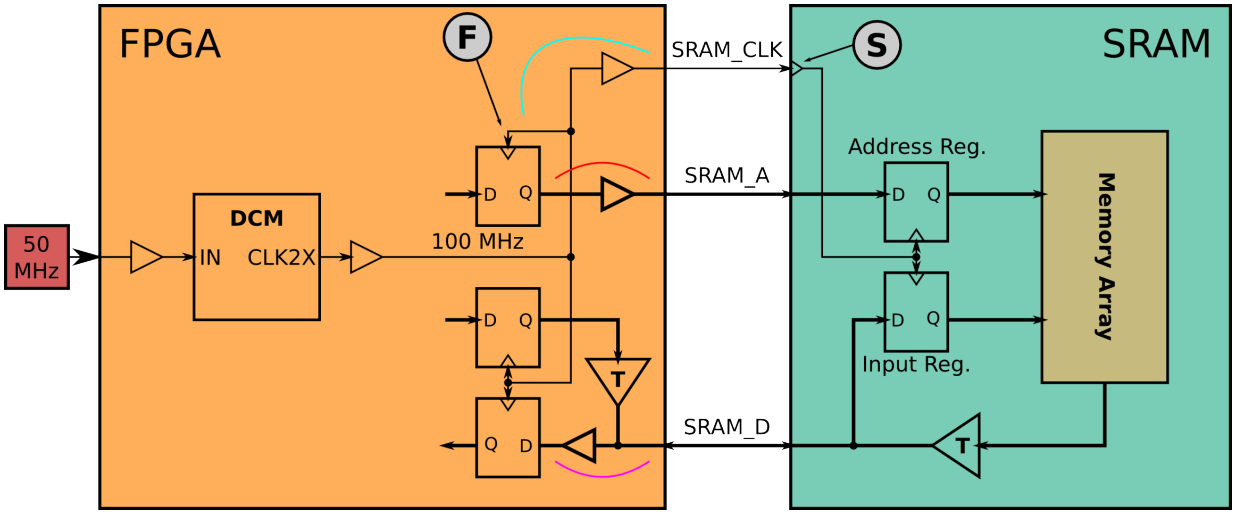

Figure 4.38: SRAM link-up (on current revisions of the PCB)

Figure 4.38 shows the (simplified) link-up of FPGA and SRAM. A DCM generates an internal 100 MHz clock signal which is fed to a global clock net of the FPGA that drives the internal registers. This signal is also routed to an output buffer that drives `SRAM_CLK` which is connected to the clock input of the SRAM. The delay on the blue path inside the FPGA is dominated by the propagation time of the buffer (T_{IOOP}) which is approx.

2.9 ns (see page 133 of [22] for reference). This results in clock skew between the registers inside the FPGA (spot F) and the registers inside the SRAM (spot S) of at least 2.9 ns. The wire on the PCB adds additional (unknown) delay to the path from F to S.

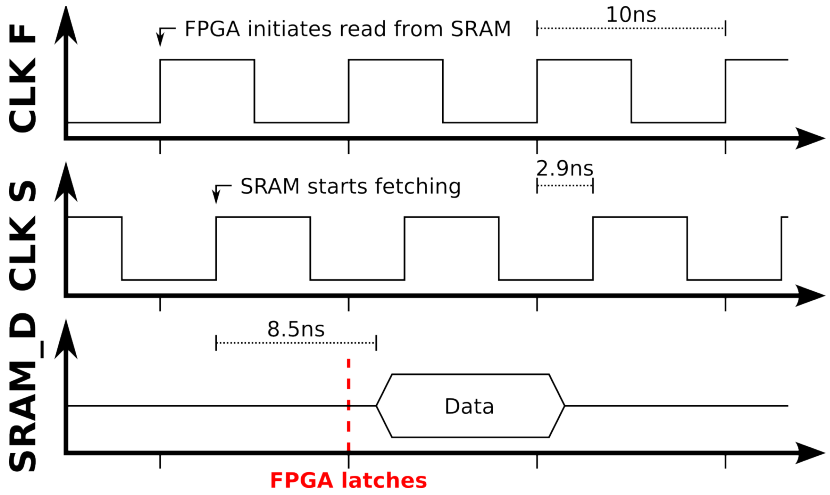

Figure 4.39: SRAM clocking (skewed)

The resulting timing is shown in figure 4.39. The FPGA initiates a read access on the rising edge of the clock driving the internal registers (CLK F). The SRAM will receive this rising edge (at least) 2.9 ns later (CLK S) and will then start fetching the data from its internal memory array. This will take approx. 8.5 ns (t_{CDV}, see page 20 of [2] for reference) and the data bus therefore will be valid (at the earliest) 11.4 ns after the rising edge (on CLK F) that initiated the read access. The data then has to propagate through the FPGA's input buffer (purple path) before it finally reaches the register in the FPGA. The FPGA however will sample the data bus (`SRAM_D`) one clock cycle (10 ns) after it has initiated the read access and therefore will latch the data bus before the SRAM has even started driving it. The value read by the FPGA thus will be invalid because of clock skew. The obvious solution seems to be a lower clock frequency or to latch the data bus not before two clock cycles have passed. Both approaches though decrease the achievable transfer rate and are prone to another problem: The path from the address register to the output driving `SRAM_A` (red) might have similar delay to the path driving `SRAM_CLK` (blue). Depending on the clock-to-output delay of the register this might lead to a hold-time violation in the SRAM. OFFSET OUT constraints (used to avoid timing violations in devices fed from the FPGA) can not be used because the toolchain only supports OFFSET constraints relative to a clock input of the FPGA.

Clock skew between two devices on a PCB is a common problem in synchronous designs and has to be eliminated as far as possible. The overall objective is that all synchronous elements in the design (e.g. registers) receive the clock edge at the same time.
In Xilinx FPGAs this can be realized using one or more DCMs. The next section shows how DCMs can be used to eliminate the clock skew inside the FPGA. Based thereon, the subsequent section (4.4.2.3) explains how external clock skew can be removed and section 4.4.2.4 describes the modification of the PCB that is required for that purpose.

4.4.2.2 Eliminating internal clock skew

In a synchronous design all registers inside the FPGA should receive the active clock edge at exactly the same time as the registers in external devices (that are clocked by the same external clock source). Therefore, the delay on the path from the FPGA's clock input pin (I) to the internal registers (F) has to be eliminated[25)].
This clock skew (between the FPGA's clock input and the internal registers) is created not only by wiring but also by components on the clock path (e.g. DCMs or buffers). The external clock signal enters the FPGA through a buffer and then is fed into the DCM (light green path in figure 4.40). Another buffer then drives the 100 MHz clock net in the FPGA that is connected to the internal registers. To eliminate the clock skew between clock input (I) and internal registers (F) all these path delays have to be known. Then a phase shift compensating the clock skew can be applied using the DCM.

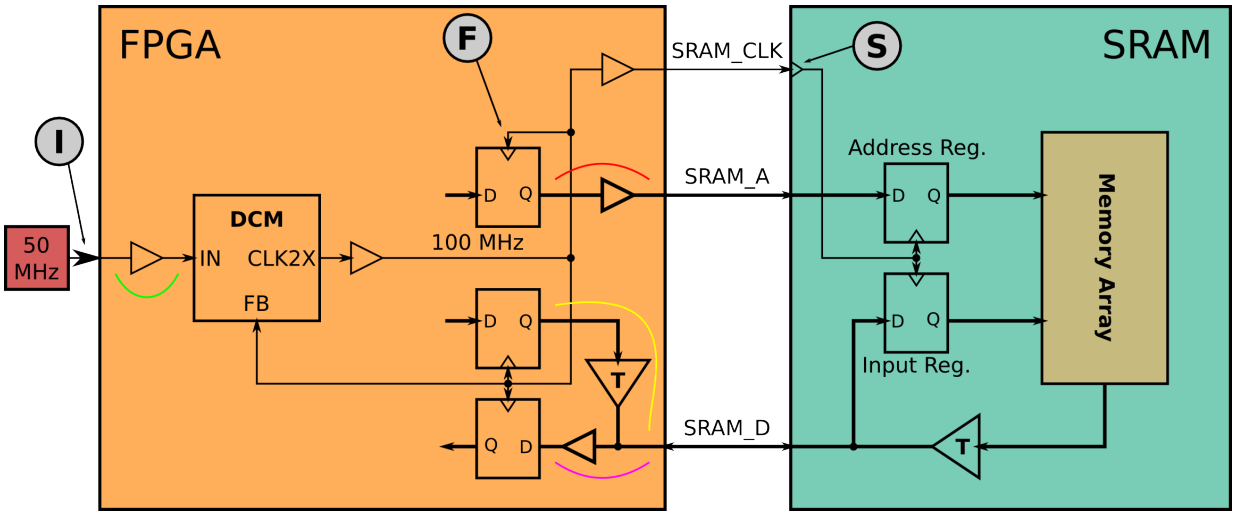

Figure 4.40: Internal skew elimination

Figure 4.40 shows the usual way to compensate internal clock skew in a Xilinx FPGA. Because the path from the clock input (spot I) to the DCM (depicted in light green) is exactly known and characterized its delay can be compensated by the DCM using a fixed phase shift[26)]. The buffer and the internal clock distribution network connected to the `CLK2X` output of the DCM however delays the clock signal before it arrives at all the registers in the FPGA. The exact delay is unknown because the internal clock network is not compensated and therefore the delay has to be measured during runtime.

[25)] Please note that on the EyeBot M6 the FPGA is the only device clocked by the external oscillator (see section 4.3.2.4 for reference). Hence it is, at first sight, unnecessary to delve into eliminating the internal clock skew (relative to the external oscillator). But because OFFSET timing constraints (needed to constrain the external SRAM) are always related to a clock input of the FPGA this topic has to be dealt with.

[26)] This is only possible for certain specific paths inside the FPGA driven by an `IBUFG` which is the case for all clock inputs of the FPGA. These paths are compensated and retain their timing characteristics even if temperature or voltage changes.

This is implemented by feeding the internal clock net back into the DCM's feedback input (`FB`). The DCM will then phase-shift its output `CLK2X` until `IN` and `FB` are in phase using a delay-locked loop control mechanism. The phase difference between the FPGA's clock input (spot I) and the internal registers (spot F) thus is controlled to be nearly zero. The registers inside the FPGA therefore will be triggered exactly at the point in time the clock edge arrives at the clock input of the FPGA.
In other words: The DCM (knowing the delay of the light green path and measuring the delay of the internal clock net) removes any phase difference between spots I and F. This compensates the delay of the internal clock net and thus eliminates the internal clock skew (between clock input pin and registers). More information about clock routing and DCMs can be found in chapters 2 and 3 of [21].

The above also explains why it is feasible to relate OFFSET timing constraints to a clock input of the FPGA instead of the clock input of the particular register. Because the DCM removes the phase shift between I and F (and therefore compensates the delay of the clock net) the OFFSET constraints can be given relative to the clock input of the FPGA. For fully synchronous systems (where all units on a PCB are clocked by the same clock source) these constraints are completely sufficient. All signals fed into the FPGA are synchronized to that external clock and all signals originating in the FPGA will be sampled synchronously to that clock (because all the other devices on the PCB are triggered by that external clock as well).
If the SRAM would be clocked by the external 50 MHz oscillator two OFFSET OUT constraints (red and yellow paths) and one OFFSET IN constraint (purple path) could be used to constrain the SRAM link-up. This however is not the case, therefore the skew between the FPGA's internal clock net and the external SRAM has to be compensated. The next section thus explains the elimination of external clock skew.

4.4.2.3 Eliminating external clock skew

External skew can be compensated in exactly the same way as internal skew. Because the delays of both the output buffer and the external clock net are not known precisely an external feedback path is needed. This feedback path is then used to determine which amount of (negative) phase shift has to be inserted to compensate the delay of output buffer and external clock net.

Figure 4.41 shows the proposed system deploying two DCMs. Just like in section 4.4.2.2 DCM1 eliminates the skew on the internal clock net of the FPGA by zeroing the resulting phase difference between the FPGA's clock input (I) and the internal clock net (F).

A compensated path connects DCM2 to the internal clock net of the FPGA (light green path). The output of the DCM is sent to the SRAM using an output buffer. A dedicated track on the PCB and another compensated path inside the FPGA (dark green) are used to provide a feedback to DCM2. DCM2 automatically compensates the delays on the two green paths using a fixed phase shift. The unknown delay of the external clock net feeding the SRAM is compensated using the feedback path. Therefore, DCM2 eliminates

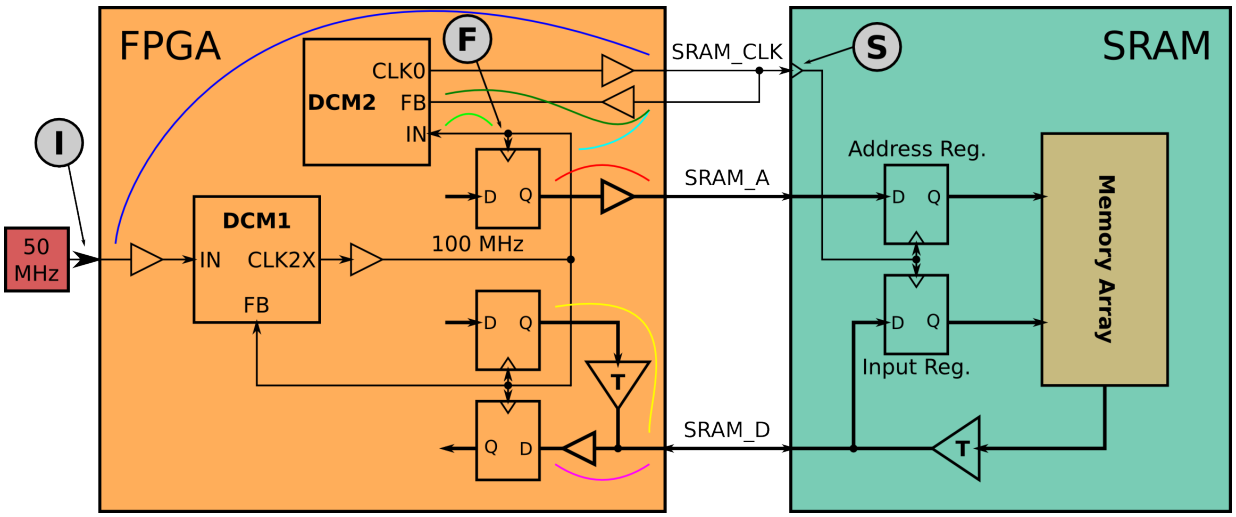

Figure 4.41: External skew elimination

any phase difference between the internal clock net (F) and the SRAM clock net (S) (light blue path). Because DCM1 at the same time eliminates any phase difference between I and F no phase difference will occur between the FPGA's clock input I and the clock net feeding the SRAM (S) (dark blue path).

In other words: The deployment of a feedback path on the PCB and two DCMs in the FPGA eliminates both internal skew in the FPGA and external skew on the clock net feeding the SRAM. A rising edge generated by the 50 MHz oscillator (I) will appear simultaneously both at the registers inside the FPGA (F) and at the SRAM (S).

This approach also solves the problem related to the timing constraints required for the external SRAM. The original problem was that the toolchain only supports timing constraints relative to (clock) inputs of the FPGA. However, a need for timing constraints relative to an output pin of the FPGA was identified earlier because one output is used to clock the external SRAM. The timing properties (setup-, hold- and clock-to-output-times) of the SRAM therefore have to be considered and calculated relative to that particular output pin of the FPGA.
Looking back at the previous findings it can be seen that the two DCMs will ensure that the clock signal sent to the SRAM is as similar as possible to the clock signal that is fed into the clock input of the FPGA. Therefore, timing constraints relative to clock inputs (which are supported by the toolchain) can be used to constrain the SRAM because no difference between the two signals can be noted anymore.

4.4.2.4 Required modifications on the PCB

Looking back at figure 4.41 it can be seen that an external feedback path is needed to deploy the approach described in section 4.4.2.3. It routes the clock signal sent to the SRAM back into the FPGA and allows the DCM to compensate the clock skew.

The current revision of the PCB however lacks this feedback path. In addition, all pins of the FPGA are currently used, therefore an input pin has to be freed first. As already mentioned in section 4.4.2.2 only certain input pins feature a compensated path to one of the four DCMs. It is recommended to use one of these input pins for the feedback path to allow for the most precise deskewing possible. More information on clock resources and DCMs can be found in chapters 2 and 3 of [21].

4.4.3 Summary

The first part of this section proposed a storage architecture that satisfies the requirements of the the stereo image processing system presented in [29]. The storage architecture provides facilities to stream data to and from the external SRAM and is tailored to the expected data flows in the resulting system. The link-up of the FPGA to both the cameras (sourcing the data into the FPGA) and the CPU (performing the final processing steps) is shown as well.
The proposed image processing system combines processing in hard- and software to achieve a higher frame rate compared to a pure software solution. Assuming a best-case scenario a maximum frame rate of 20 frames/s was found for the processing steps implemented in hardware. The actual frame rate depending on the exact timing relation between cameras, FPGA and CPU will be lower and has to be determined by measuring.

The actual implementation of the storage architecture however was rendered impossible by a previously unnoticed problem on hardware-level (related both to timing constraints and clock skew on board-level). The second part of this section therefore analyzed this problem, presented the general approach to eliminate clock skew and proposed a solution that can be incorporated in the next revision of the PCB.

It should be mentioned that a basic SRAM controller was implemented to allow for timing analyses and measurements. More information can be found in appendix E.

5 Conclusion and future work

The thesis at hand presented the communication and image processing framework that was developed for the EyeBot M6. The framework is designated to be loaded into the FPGA and was specifically tailored to fit the requirements of the platform. It solves the data transfer errors between CPU and FPGA observed in previous designs, introduced a storage architecture suitable for image processing systems and forms a foundation for further development.

The work was partitioned into four major parts:
First of all, the old existing design was analyzed and the reasons for the transfer errors were exposed. The asynchronous interface between CPU and FPGA was found to be responsible and had to be redesigned from scratch. In addition, it was noticed that an important step in the FPGA design-flow (that ensures operativeness if the FPGA design grows) has been skipped.
To solve the transfer errors several alternative approaches to interface CPU and FPGA have been investigated. Several interfacing methods have been deduced and checked for applicability. Due to platform limitations it was impossible to replace the asynchronous interface which, if possible, would have simplified both system and development. Nevertheless two reasonable approaches were found, implemented and deployed successfully. Additionally, the impact of the new interface on the achievable transfer speed between CPU and FPGA has been evaluated. A part of the existing code base has been ported to the new design to facilitate backward compatibility with the old design.
Subsequently the missing step in the FPGA design-flow was performed to ensure the continuous operativeness of the new design. The effectuality of the measures taken was verified both using simulation and performing measurements on the hardware.
Furthermore, a storage architecture was designed that facilitates access to the external SRAM. The SRAM was required by the image processing system developed by a fellow student that pursues the generation of a depth map. Its requirements were analyzed and used to tailor the storage architecture to the expected data flows. The link-up between the storage architecture and the CPU was taken care of, too. Based thereon, the achievable frame rate of the image processing system (using the storage architecture) was estimated. A problem on the PCB however precluded the use of the SRAM and had to be investigated. A solution to the problem has been found but could not be deployed because a time-consuming redesign of the PCB is required. The proposed storage architecture can come into operation at a later point in time.

The modification related to the SRAM is not the only change that has to be included in the new revision of the PCB. In addition, the following problems still have to be solved:

The camera interface is unstable if the cameras are switched to the color mode required by the image processing system. Furthermore, the current link-up of the external clock sources to the FPGA's global clock inputs is disadvantageous. This not only complicates the clock routing inside the FPGA but also impedes the utilization of more than one internal system clock. Finally, the existing VHDL module used for servo control (PWM generation) unnecessarily wastes precious logic resources and thus should be revised.
More detailed suggestions that should be considered for the next revision of the PCB can be found in appendix F. It is essential that these low-level flaws are fixed before further time is spent on high-level tasks (that depend on the maturity of the lower level).

A VLIO timing values

During research various sources of information regarding the timing of the VLIO interface have been considered. Chapter 6.7.6 (page 6-55) of [5] (Developers Manual) describes the interface and gives two waveforms, one for a read-burst and one for a write-burst. Chapter 2.6.5 (page 2-10) of [4] (Design Guide, DG) presents a similar description, one waveform and a table of timing parameters (in ns). Finally, the same timing parameters (this time in MEMCLKs) can be found in chapter 4.8 (page 35) of [6] (Specification, Spec.). Inconsistent values are marked with "!!", the ones chosen for simulation and constraining the timing are the ones from the Developers Manual.

In addition, two data sheets of the PXA270 ([7] and [8]) were used as reference (because they are more precise than the ones for the PXA255). Changes between PXA255 (Developers Manual) and the PXA270 are indicated by "¡¡".

Table A.1 summarizes all sources. All values are given as multiple of MEMCLK (= 10 ns).

The timing parameters were set to following values: RDF=3, RDN=2 and RRR=1.

name	description	value valid for	PXA255 Developers Manual	PXA255 DG & Spec.	PXA270 (AC Timing) Specification	255 = 270
t_{AS}	address setup to $\overline{\text{CS}}$ asserted	W, R	1	1	1	y
t_{AH}	address hold after $\overline{\text{OE}}$ or $\overline{\text{PWE}}$ deasserted	W, R	1	1	¡¡ RDN ¡¡	n
t_{ASRW0}	address setup to $\overline{\text{OE}}$ or $\overline{\text{PWE}}$ asserted (1st beat)	W, R	3	?	3	y
t_{ASRWn}	address setup to $\overline{\text{OE}}$ or $\overline{\text{PWE}}$ asserted (following)	W, R	RDN	!! 1 !!	RDN	y
t_{CES}	$\overline{\text{CS}}$ setup to $\overline{\text{OE}}$ or $\overline{\text{PWE}}$ asserted	W, R	2	2	2	y
t_{CEH}	$\overline{\text{CS}}$ hold after $\overline{\text{OE}}$ or $\overline{\text{PWE}}$ deasserted	W, R	1	1	1	y
t_{DSWH}	minimum write data setup to $\overline{\text{PWE}}$ deasserted	W	RDF+2	!! 2 !!	RDF+2	y
t_{DH}	data hold after $\overline{\text{PWE}}$ deasserted (t_{DHW})	W	1	1	1	y
t_{DSOH}	data setup to "address changing"	R	1.5	?	1.5	y
t_{DOH}	data hold after "address changing" (t_{DHR})	R	0	0	0	y
t_{RDYH}	RDY hold after $\overline{\text{OE}}$ or $\overline{\text{PWE}}$ deasserted	W, R	0	0	0	y
t_{RWA}	$\overline{\text{OE}}$ or $\overline{\text{PWE}}$ assert time	W, R	RDF+1+waits	?	RDF+1+waits	y
t_{RWD}	$\overline{\text{OE}}$ or PWE deassert time (t_{NPWE})	W, R	RDN+2	!! 2 !!	¡¡ 2×RDN ¡¡	n
t_{CD}	minimum $\overline{\text{CS}}$ deassert time between two transfers	W, R	2×RRR+1	?	2×RRR+1	y

Table A.1: Timing of the VLIO interface

Note: Chapter 6.7.6 (page 6-55) of [5] states "Data will be latched on the rising edge of MEMCLK once the internal RDY signal is high and the minimum assertion time of

RDF+1 has been reached" and "The nOE or nPWE signal will de-assert one MEMCLK after data is latched". Considering the minimum assertion time of $\overline{\text{OE}}$ or $\overline{\text{PWE}}$ being RDF+1, the first quote should state "Data will be latched on the rising edge of MEMCLK once the internal RDY signal is high and RDF MEMCLKs have passed by".
Additionally, "This means that for a transaction to complete at the minimum assertion time for either nOE or nPWE (RDF+1), the RDY signal must be high two clocks prior to the minimum assertion time for either nOE or nPWE(RDF-1)" should state "(...) the internal RDY signal must be high (...)".

B FIFO-based approach to interface CPU and FPGA using VLIO if CPU clock is available

The FIFO-based approach described in section 4.2.1.2 was using the SRAM-interface of the CPU's memory controller. Therefore, a clock signal was available to drive the FIFO-ports turned towards the CPU. To implement write accesses the write-port of FIFO1 was connected to the VLIO-interface (CPU-CD) and the read-port was part of the FPGA-CD. For read accesses the read-port of FIFO2 was connected to the VLIO-interface (CPU-CD) and the write-port was part of the FPGA-CD (see figure 4.2 for reference). All problems related to CDCs would have been handled by the independent clock FIFOs internally.
Though a limitation of this approach has been identified already: Because read-accesses have to traverse through both FIFOs additional clock cycles (compared to the VLIO-setup depicted in figure 3.2 and figure 3.3) were needed which lead to a slowdown of both read- and write-accesses.

This approach could be modified by using the VLIO-interface instead of the SRAM-interface and still keeping a clock signal from the CPU wired to the FPGA.
Because we assume the availability of a clock signal from the CPU all CDC-related problems can be solved as in the previous approach (if the design is done carefully).
Additionally, it would be possible to speed up write-accesses by using the RDY-pin (see section 4.2.5.1) of the VLIO-interface to produce the additional delay needed for read-accesses: The EMPTY status flag of FIFO2 could be used as indicator whether the requested data has already arrived in the CPU-CD. As long as a read-request is still processed by the FPGA FIFO2 will be empty, its EMPTY signal will be asserted and RDY will be deasserted by the interface-unit inside the FPGA. If the requested data is finally available in FIFO2 EMPTY will be deasserted. This deassertion can then be used to assert RDY which informs the CPU that the data bus now contains valid data. Therefore, a read-access will be lengthened by exactly the time needed to fetch the data.
If the CPU commences a write-transaction the FPGA will assert RDY immediately (if FIFO1 is not full) because FIFO1 can easily store the address and the data supplied by the CPU and no wait states are needed. Therefore, write-accesses will not be slowed down.
In other words: During read-accesses RDY will be connected to the negated EMPTY signal of FIFO2, during write-accesses it will be connected to the negated FULL signal of FIFO1. This flow control therefore implicitly also takes care of avoiding over- and underflows in both FIFOs.

This approach therefore has the following advantages over the approach described in section 4.2.1.2: Write-transfers will run at the highest speed supported by the bus and read-transfers will only be slowed down as much as needed for the (dual) CDC. Overflows in FIFO1 and underflows in FIFO2 will never happen because the status signals of the FIFOs are used to stop the CPU until the respective error condition has been resolved.

Looking back at section 4.2.4 it can be found that continuous clocks are required on both ports of an independent clock FIFO. They are used to drive the two synchronizers inside the FIFO that are involved in the updating of the status flags (FULL and EMPTY).
If the clock is non-continuous a deadlock could arise: During a read-request the RDY-pin will be driven with the negated value of FIFO2's EMPTY. Therefore, the CPU will stop until EMPTY gets deasserted by the FIFO which will only happen if two conditions are fulfilled: First some data has to be written to the write-port of FIFO2 (in the FPGA-CD) and second a few clock cycles on the read-port are needed to synchronize this change on the write-port side to the read-port side. If the read-clock is not running at this point in time RDY will never be asserted (because EMPTY will never get deasserted) and the CPU will wait infinitely.

Again this approach can not be used on the EyeBot M6 because there is no clock signal leaving the gumstix that could be connected to the FPGA.

C Additional VLIO waveforms

This section presents additional waveforms of the VLIO transfer qualifiers. Section 4.2.7.4 already showed MMIO- (and therefore single-beat-)accesses. Again channel 1 is connected to $\overline{\text{CS}}$, channel 2 to $\overline{\text{PWE}}$ resp. $\overline{\text{OE}}$ and channel 3 to RDY. The signals were routed to the camera-port to measure the waveforms because it is impossible to reach the signals on the PCB due to the lack of test-pads. Thus, there might be a (small) phase offset between the different traces depending on the routing inside the FPGA.

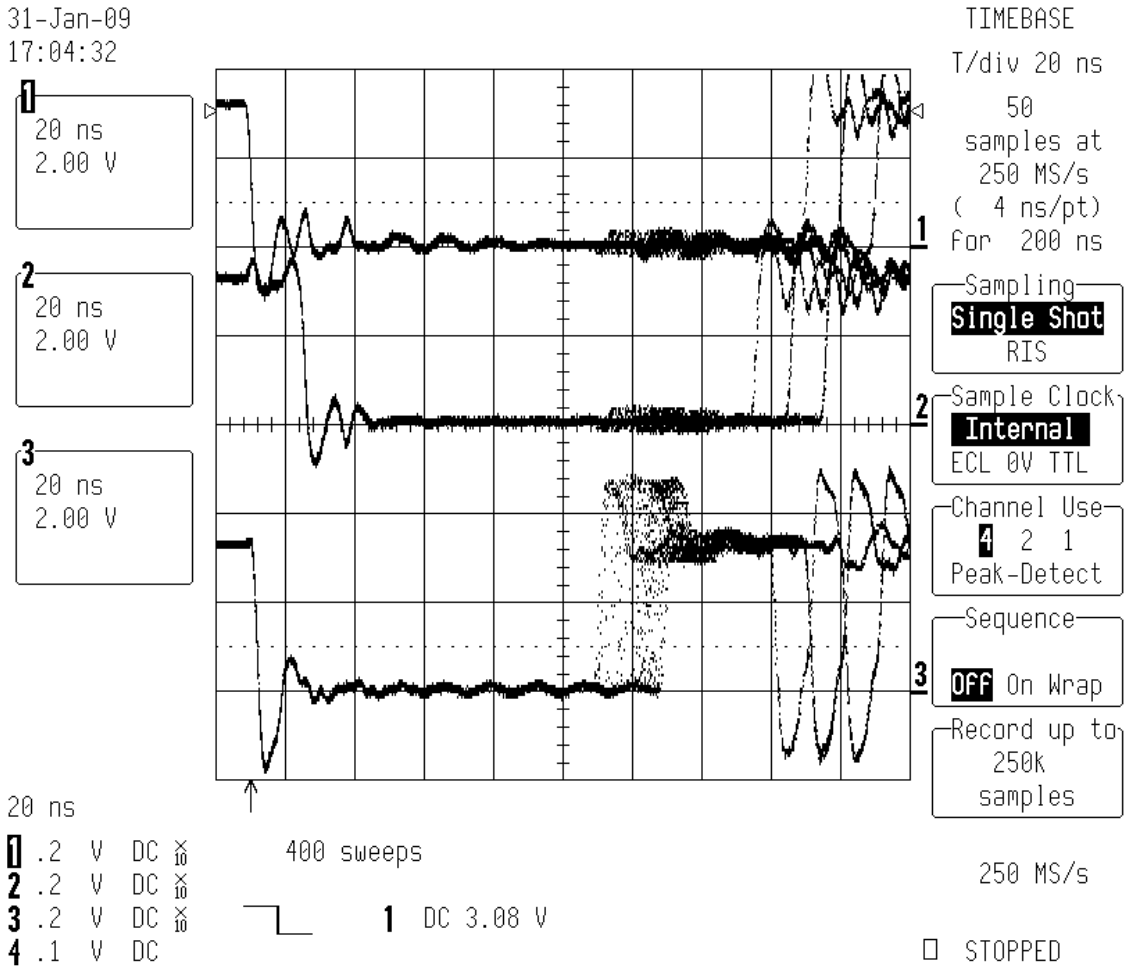

Figure C.1: PB read using MMIO (sweep with 400 samples)

Figure C.1 shows a sweep of 400 VLIO reads from the PB using MMIO. The trigger (again) is set to the asserting (falling) edge of $\overline{\text{CS}}$. It can be seen that the duration of a

read access is not constant because the deasserting (rising) edge of each particular signal happens at different points in time after the asserting (falling) edge. This behaviour was expected because there are two synchronizers involved (one transferring `X_PB_RD` from CPU- to FPGA-CD and one inside the PXA-CPU transferring RDY from the FPGA- to the CPU-CD). Each of this synchronizers introduces an uncertain delay approx. equal to the clock period (see section 4.2.3.1 for reference).
The RDY signal gets asserted [100; 120] ns after $\overline{\text{CS}}$ has been asserted. It is not understood yet why the rising edge of RDY is somehow distributed in such an interval. The author expected two possible points in time for this rising edge, one at 100 ns and one at 120 ns after the assertion of $\overline{\text{CS}}$. The reason therefore is that RDY is asserted synchronously to the FPGA-clock (that has a period of 20 ns) and this assertion only depends on one synchronizer (the one that transfers `X_PB_RD`) which has an uncertainty of 20 ns.
The deasserting edge of $\overline{\text{CS}}$ happens at 160 ns, 170 ns or 180 ns after the assertion of $\overline{\text{CS}}$. This additional uncertainty is created by the synchronizer transferring RDY to the CPU-CD that runs at a frequency of 100 MHz (which is equivalent to a period of 10 ns). The case of an $\overline{\text{CS}}$ assertion time of 170 ns is the one already depicted in figure 4.29.

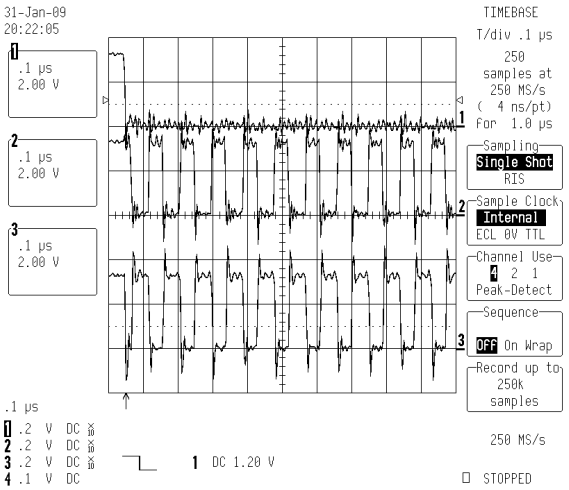

Figure C.2: PB write using VLIO-bursts

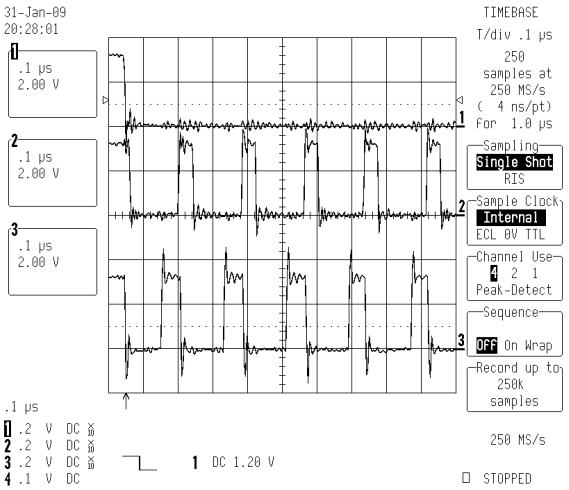

Figure C.3: BB read using VLIO-bursts

Figure C.2 shows a write to the PB using VLIO-bursts (generated by a DMA-transfer) similar to figure 4.21. Identical to the single-beat access depicted in figure 4.27 the RDY-pin is only deasserted until $\overline{\text{PWE}}$ gets asserted. Figure C.3 shows a read from the PB using VLIO-bursts (generated by a DMA-transfer) similar to figure 4.22. Again the RDY-pin is used during each single beat to throttle the CPU identical to the single-beat read depicted in figure 4.29.

Waveforms of DMA transfers to resp. from the BB look exactly as depicted in figure 4.21 resp. figure 4.22 and thus are not shown.

D VHDL entities, component hierarchy and additional data types

The following gives an overview of the deployed VHDL-entities, their functions and the resulting hierarchy. Additional data types used to increase readability are presented, too.

Table D.1 lists all user-defined VHDL data types used in this design. They are defined in `system_pack` which therefore has to be included (`USE`d) in every entity. Some of them can be modified using constants, e.g. the address types used to split the address bus. Helper functions ease the splitting of the address bus into its sub-parts.

Name	Width	Description
`data_t`	16	generic data type (used for `datamux`)
`data_array_t`	? × 16	unconstrained array of `data_t`
`unknown_addr_t`	?	address of unknown width
`cpu_addr_t`	20	CPU address bus
`cpu_data_t`	16	identical to `data_t`
`cpu_data_array_t`	? × 16	unconstrained array of `cpu_data_t`
`fpga_data_t`	16	identical to `data_t`
`fpga_addr_t`	20	address bus inside FPGA, split into:
`fpga_addrhigh_t`	2	MSBs of address bus to select bus system
`fpga_addrlow_t`	18	remainder of address bus
`fpga_pb_data_t`	16	identical to `fpga_data_t`
`fpga_pb_addr_t`	18	identical to `fpga_addrlow_t`, split into:
`fpga_pb_modaddr_t`	8	selects device on PB
`fpga_pb_subaddr_t`	10	selects register inside device on PB
`fpga_pb_enable_t`	7	enables for PB (one-hot encoded)

Table D.1: VHDL data types

Table D.2 lists all entities of the design and their function and states where each respective entity is instantiated. Entity-names generally end with `_entity` to ease the distinction between entity and instantiated component. To shorten the synthesis' runtime and to free FPGA resources a configuration constant allows the disabling of the old m6-modules.

Name of entity	instantiated in	Description/Reference
Supporting units		
address_decoder	bus_bridge, bus_bridge_pb, dataregfile	decodes address to one-hot
bitmux	bus_bridge	multiplexer (1 bit signal)
datamux	bus_bridge, dataregfile, pb_subsys, toplevel	multiplexer (16 bit bus)
datareg	dataregfile	16 bit register
dataregfile	bus_bridge, hardware_vision, sram_controller	scalable register-file
CPU interface		
bus_bridge	toplevel	4.2.7
bus_bridge_pb	bus_bridge	4.2.7.2
bus_bridge_testregs	bus_bridge	debug-module
Feature units		
hardware_vision	toplevel	see [29]
pb_subsys	toplevel	instantiates old m6-modules
m6xdio	pb_subsys	digital I/O
m6xdrv	pb_subsys	registers for motor-control
m6xmot	m6xdrv	PWM/encoder (single motor)
m6xsen	pb_subsys	registers for PSDs
m6xpsd	m6xsen	data acquisition (single PSD)
m6xhnd	pb_subsys	Registers for servos
m6xsrv	m6xhnd	PWM (for single servo)
sram_controller	toplevel	4.4.3
Toplevel		
toplevel	(toplevel_tb)	4.2.8
toplevel_tb	–	testbench (4.2.7.5)
Xilinx cores		
m6clk_dcm	toplevel	generates internal clocks
fifo_bram_18x1k	bus_bridge	independent clock FIFO
Supporting packages		
component_pack	–	components
component_pack_coregen	–	CoreGen components
misc_mgeier_pack	–	testbench helpers
system_pack	–	data type definitions

Table D.2: VHDL entities (without _entity-suffix)

E Address map of the design

Table E.1 shows the address map of the VHDL design. More detailed information can be found in `fpga_busbridge.h`. This file contains a C-macro to generate the start address of each module on the PB (`GETPBBASE()`) and various constants to access BB (`BBBASE`) and `csreg`.

start address	end address	bus/module	function
0x00000	0x3FFFF	PB	
0x00000	0x003FF	m6xdio	Digital I/O[1)]
0x00400	0x007FF	m6xdrv	Motor control[1)]
0x00800	0x00BFF	m6xsen	PSD acquisition[1)]
0x00C00	0x00FFF	m6xhnd	Servo control[1)]
0x01000	0x013FF	sram_controller	SRAM interface
			see table E.2
0x01400	0x017FF	hardware_vision	Hardware vision
0x01800	0x01BFF	(in toplevel)	address-mirror
0x01C00	0x3FFFF	unused	
0x40000	0x7FFFF	BB	FIFO write and read
0x80000	0xBFFFF	csreg	FIFO-control
0x80000			ID register (set to 0xBB)
0x80001			REGIN1: debug[2)]
0x80002			REGIN2: fpga2cpu-FIFO[3)]
0x80003			REGIN3: cpu2fpga-FIFO[4)]
0x80004			REGOUT1: debug[5)]
0xC0000	0xFFFFF	debug	counts write- and read-accesses

Table E.1: Address map

Table E.2 shows the register map of the basic SRAM controller used for timing analyses and measurements.

[1)] See `libM6fpga/libM6fpga.h` in the library source for detailed information on the single registers.
[2)] Bit 0: TMPIN
[3)] Bits 9-0: rdcnt, bit 14: empty, bit 15: valid
[4)] Bits 9-0: wrcnt, bit 14: almost full, bit 15: full
[5)] Bit 0: TMPOUT

address	name	description
0x01000	RDDATAREG1	SRAM_D[15:0] after read
0x01001	RDDATAREG2	0x1230 & SRAM_D[17:16] after read
0x01002	CTRLREG1	Bit 0 (write): Set to 1 to initiate a write access Bit 1 (read): Set to 1 to initiate a read access Both bits are reset automatically
0x01003	CTRLREG2	Bit 0 (enable): If set to 0 SRAM_PWROFF is asserted
0x01004	ADDRREG1	driven to SRAM_A[15:0]
0x01005	ADDRREG2	Bits 3 to 0 are driven to SRAM_A[19:16]
0x01006	WRDATAREG1	driven to SRAM_D[15:0] during write
0x01007	WRDATAREG2	Bits 1 to 0 are driven to SRAM_D[17:16] during write

Table E.2: Register map

F Suggestions for improvement

F.1 Camera link-up

As already mentioned in section 3.5 the old camera interface was prone to timing violations, too. An independent clock FIFO was used to transfer the signals from the camera-CD to the FPGA-CD (see figure 4.37, section 4.4.1.2, for reference). This approach eliminated the pixel errors observed previously. The FIFOs did not only facilitate the CDC but also served as buffer to collect enough data for a BB-transaction.
Though a problem with this approach was found: The control signals generated by the camera (VSYN and HREF) could not be detected reliably when using the default frame rate of 50 frames/s. Only if the camera was slowed down (using its internal prescaler) a stable image could be received. The probability of success depended both on the prescaler value and on the drive strength setting of the camera's output buffers.
An electrical problem thus seems to be responsible for the observed errors and needs further attention. Rise- and fall-times and voltage levels of the camera signals should be checked as close as possible to the FPGA.

The following commands are needed to reproduce the problem:

```
# disable both cameras
echo set > /proc/gpio-ac97/UCB1400-0-4
echo set > /proc/gpio-ac97/UCB1400-0-5
# enable cam1
echo clear > /proc/gpio-ac97/UCB1400-0-4
# initiate software reset
i2c wb 0x60 0x12 0x80

# switch to YUV mode
i2c wb 0x60 0x12 0x34
i2c wb 0x60 0x13 0x21
# increase drive strength
i2c wb 0x60 0x20 0x01
i2c wb 0x60 0x38 0xc1
```

This will result in a misaligned or garbled picture. To slow down the camera the following command can be used:

```
i2c wb 0x60 0x11 0x08
```

In addition, it is advisable to route both camera clocks (CAM1_PCLK and CAM2_PCLK) to dedicated clock input pins. See section F.2 for additional information.

If possible though the cameras should be synchronized to the FPGA using the "slave mode" of the camera module. This has not been investigated further.

F.2 Clock input allocation

When reallocating the clock inputs of the FPGA (which is required for the SRAM feedback path, see section 4.4.2.4) the following has to be taken into consideration:

- The FPGA is partitioned into four clock regions that all receive all global clocks.
- Eight global clock nets can be driven by one of the 16 global clock inputs or a signal derived therefrom (using a DCM).
- Four of these eight global clock nets each originate in the upper resp. lower half of the device.
- Additional clock nets are available in the left resp. right half of the device only.
- Because only four clock signals can be injected per half it is advisable to route
  - M6CLK to an input pin connected to the upper half and to route
  - $\overline{\text{PWE}}$ and $\overline{\text{OE}}$ to an input pin assigned to the lower half.

  This ensures that both $\overline{\text{PWE}}$ and $\overline{\text{OE}}$ and a maximum of four clock signals derived from M6CLK[1)] can be routed to a global clock network (and therefore are available everywhere in the FPGA). The SRAM clock feedback can be connected to a (global) clock input pin that is associated with an unused DCM, preferably one in the lower half (because two more global clock nets can be injected from that half).
- Additional clocks (e.g. CAM1_PCLK and CAM2_PCLK) should be routed to clock inputs of left resp. right half because they are only used to clock a small amount of logic (and an independent clock FIFO).

F.3 CPU link-up

If anyhow possible a clock signal from the CPU should be routed to the FPGA. This would not only further reduce the probability of metastability issues but also allow the usage of the faster S(D)RAM interface.

[1)] With the current clock input assignment a maximum of two clocks derived from M6CLK can be routed to a global clock network because two (out of four) global clock nets (originating in that half) are required for $\overline{\text{PWE}}$ resp. $\overline{\text{OE}}$. If the second DCM in the upper half of the device is used to generate the SRAM clock only one global clock net is left for a clock signal derived from M6CLK.

In addition, the buffers on address- and data-bus introduce additional delay. The transfer qualifiers ($\overline{\text{PWE}}$, $\overline{\text{OE}}$ and RDY) however are unbuffered and therefore undelayed.
This was factored into the timing constraints presented in section 4.3 and should not impose a problem anymore. If however (at a later stage of development) the toolchain fails to fulfil the timing constraints some additional slack could be gained by removing the buffers and adapting the constraints.

F.4 JTAG link-up and boundary scan

Some of the ICs on the PCB (FPGA, SRAM and gumstix) supports boundary scan testing (BST) via the JTAG interface. BST can be used to verify the solder joints on an assembled PCB. Devices that support BST contain additional logic that allows to write to and read from (nearly) all pins of the device. Given the case that two devices supporting BST are connected to a PCB track its connectivity can be verified easily: One IC drives known values to the track and the values received by the other IC are read back. If the values match the track and all associated solder joints are considered working.
In addition, the link-up of the FPGA's JTAG interface allows the use of the FPGA's debugging capabilities.
Multiple devices can be served by one single JTAG interface using a chain-based wiring.

F.5 General purpose I/Os and PWM signals

Several I/O-pins of the FPGA are used for "low-speed" purposes. Fourteen pins are needed for servos and 8 pins are occupied for motor control. In addition, four pins are used as general purpose digital I/O.
The PWM signal generation currently is implemented in an inefficient way that consumes a lot of FPGA resources. Instead of rewriting the VHDL code it seems reasonable to relocate the generation of the PWM signals from the FPGA to an external microcontroller. The microcontroller could be connected to the CPU's I2C-bus already used to control the cameras. It therefore would not consume any additional I/O resources on the board. Presumably the microcontroller would be powerful enough to handle additional general purpose I/O as well. The microcontroller could be hooked up to the JTAG chain already proposed above to allow for easy programming.

This would free not only precious logic resources but also 26 I/O-pins that then could be used to widen the data bus between FPGA and CPU to 32 bit (which would double the achievable transfer rate).

Bibliography

[1] *Buildroot (homepage).* http://buildroot.uclibc.org. 6

[2] *Cypress "18-Mbit (512K x 36/1M x 18) Flow-Through SRAM" (PDF).* http://download.cypress.com.edgesuite.net/design_resources/datasheets/contents/cy7c1383d_8.pdf. 92

[3] *gumstix inc. (homepage).* http://www.gumstix.com. 5

[4] *"PXA255 Processor Design Guide [278694-001]" (from gumstix.org).* http://pubs.gumstix.org/documents/PXA%20Documentation/PXA255/PXA255%20Processor%20Design%20Guide%20%5b278694-001%5d.pdf. 99

[5] *"PXA255 Processor Developers Manual [278693-002]" (from gumstix.org).* http://pubs.gumstix.org/documents/PXA%20Documentation/PXA255/PXA255%20Processor%20Developers%20Manual%20%5b278693-002%5d.pdf. 5, 22, 99

[6] *"PXA255 Processor Electrical, Mechanical, and Thermal Specification [278805-002]" (from gumstix.org).* http://pubs.gumstix.org/documents/PXA%20Documentation/PXA255/PXA255%20Electrical,%20Mechanical,%20and%20Thermal%20Spec%20%5b278805-002%5d.pdf. 99

[7] *"PXA270 Processor Developers Manual [280000-002]" (from gumstix.org).* http://pubs.gumstix.org/documents/PXA%20Documentation/PXA270/PXA270%20Processor%20Developer's%20Manual%20%5b280000-002%5d.pdf. 99

[8] *"PXA270 Processor Electrical, Mechanical, and Thermal Specification [280002-005]" (from gumstix.org).* http://pubs.gumstix.org/documents/PXA%20Documentation/PXA270/PXA270%20Electrical,%20Mechanical,%20and%20Thermal%20Specification%20%5b280002-005%5d.pdf. 99

[9] *"RoBIOS Library Functions" (not up-to-date).* http://robotics.ee.uwa.edu.au/eyebot/doc/API/library.html. 3

[10] *Usenet post of Austin Lesea, subject "Avoiding meta stability? Finally...? Don't use SRL16 as a synchronizer", MsgID di464l$q3s9@xco-news.xilinx.com (via Google).* http://groups.google.com.au/group/comp.arch.fpga/msg/9dc08821c88487a6?dmode=source. 77

[11] *Usenet post of Brian Philofsky, subject "Re: Metastablility", MsgID c4km0t$dt73@xco-news.xilinx.com (via Google).* http://groups.google.com.au/group/comp.arch.fpga/msg/736207f4d71408ee?dmode=source. 28, 31

[12] *Usenet post of Peter Alfke, subject "Re: Asynchronous FIFO and almost empty - bug?", MsgID d0944301-6d2e-4a13-a795-a79df3b00b91@b40g2000prf.google groups.com (via Google).* http://groups.google.com.au/group/comp.arch.fpga/msg/e345874a7b31b7ac? 46

[13] *Usenet post of Peter Alfke, subject "Re: Avoiding meta stability?", MsgID 11291361 10.912573.207750@g14g2000cwa.googlegroups.com (via Google).* http://groups.google.com.au/group/comp.arch.fpga/msg/9200599cab27beba?dmode=source. 28

[14] *Usenet post of Peter Alfke, subject "Re: dual clock fifo", MsgID 3069a31f-755f-4a46-ad74-fc99d7ce1292@s13g2000prd.googlegroups.com (via Google).* http://groups.google.com.au/group/comp.arch.fpga/msg/2ce9a8b8c24a5f3b? 46

[15] *Usenet post of Peter Alfke, subject "Re: Metastablility", MsgID BC9062CA.5D 26%peter@xilinx.com (via Google).* http://groups.google.com.au/group/comp.arch.fpga/msg/13f0fd611ca74381?dmode=source. 31

[16] *Xilinx Application Note 094 "Metastable Recovery in Virtex-II Pro FPGAs".* http://www.xilinx.com/support/documentation/application_notes/xapp094.pdf. 28, 29, 30

[17] *Xilinx "'Constraints Guide"' (PDF).* http://toolbox.xilinx.com/docsan/xilinx10/books/docs/cgd/cgd.pdf. 41, 70, 71, 72, 73

[18] *Xilinx PLD Blog: Post of Peter Alfke, subject "Metastable Delay in Virtex FPGAs", dated "03-28-2008 03:18 PM".* http://forums.xilinx.com/t5/PLD-Blog/Metastable-Delay-in-Virtex-FPGAs/bc-p/8165. 28, 29

[19] *Xilinx "FIFO Generator".* http://www.xilinx.com/support/documentation/ipmeminterfacestorelement_fifo_fifogenerator.htm. 46

[20] *Xilinx "FIFO Generator v4.4 User Guide" (PDF).* http://www.xilinx.com/support/documentation/ip_documentation/fifo_generator_ug175.pdf. 74

[21] *Xilinx "Spartan-3 Generation FPGA User Guide" (PDF).* http://www.xilinx.com/support/documentation/user_guides/ug331.pdf. 43, 94, 96

[22] *Xilinx "Spartan-3E FPGA Family Data Sheet" (PDF).* http://www.xilinx.com/support/documentation/data_sheets/ds312.pdf. 92

[23] *Xilinx "Synthesis and Simulation Design Guide" (PDF).* http://toolbox.xilinx.com/docsan/xilinx10/books/docs/sim/sim.pdf. 8

[24] *Xilinx "XST User Guide" (PDF).* http://toolbox.xilinx.com/docsan/xilinx10/books/docs/xst/xst.pdf. 8

[25] Blackham, Bernard: *The Development of a Hardware Platform for Real-time Image Processing*, October 2006. http://robotics.ee.uwa.edu.au/theses/2006-Embedded-Blackham.pdf. 3, 4, 52

[26] Cummings, Clifford E.: *Synthesis and Scripting Techniques for Designing Multi-Asynchronous Clock Designs*, Mar 2001. `http://www.sunburst-design.com/papers/CummingsSNUG2001SJ_AsyncClk.pdf`. 37

[27] Cummings, Clifford E.: *Simulation and Synthesis Techniques for Asynchronous FIFO Design*, Apr 2002. `http://www.sunburst-design.com/papers/CummingsSNUG2002SJ_FIFO1.pdf`. 45

[28] Cummings, Clifford E. and Peter Alfke: *Simulation and Synthesis Techniques for Asynchronous FIFO Design with Asynchronous Pointer Comparisons*, Apr 2002. `http://www.sunburst-design.com/papers/CummingsSNUG2002SJ_FIFO2.pdf`. 46

[29] Dietrich, Benedikt: *Design and Implementation of an FPGA-based Stereo Vision System for the EyeBot M6*, February 2009. t6, 21, 68, 79, 82, 90, 96, 106

[30] Gray, Frank: *Pulse Code Communication (U.S. Patent 2,632,058)*, filed 1947, granted 1953. `http://patft.uspto.gov/netacgi/nph-Parser?patentnumber=2632058` (click on "Images" to browse the document). 39

[31] Pfister, Andrea: *Metastability in digital circuits with emphasis on CMOS technology*, 1989. 29, 30

[32] Rabaey, Jan M., Anantha Chandrakasan and Borivoje Nikolic: *Digital Integrated Circuits – A design perspective (2nd Edition)*, 2003. 15, 30, 31