Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Development of Intelligent Camera Components

MSc Thesis

Author
Balázs Mészáros

Consultants
György Orosz
Károly Molnár (external)

14th December 2012

Contents

Kivonat

Diplomatervem az EDICAM (Event Detection Intelligent Camera) néven ismert intelligens kamera rendszer fejlesztésébe nyújt betekintést. A dolgozatban a két FPGA-n megvalósított firmware tervezése és implementálása kerül bemutatásra.

Az EDICAM rendszer hardver komponense egy kamerafejből és egy fejlesztői kártyából épül fel. Ezen egységek már a fejlesztés első fázisában rendelkezésre álltak. A diplomaterv első fejezete ezeket a komponenseket mutatja be az újrafelhasznált, külső fejlesztők által biztosított FPGA-modulokkal kiegészítve. Az újrafelhasznált modulok a firmware-hez kapcsolódó alacsony szintű interfészek kezelését látják el. A felhasznált komponenseken túl az FPGA-tevezéskor alkalmazott speciális időzítési megfontolások is a dolgozat elején kerülnek bemutatásra.

A firmware-fejlesztés alapjául az EDICAM rendszer magasszintű specifikációja szolgált. A dolgozat második része a specifikáció bizonyos részeit mutatja be az expozícióval és a képkiolvasással kapcsolatos megfontolásokra fókuszálva.

A specifikáció bemutatását követően a rendszertervezéssel kapcsolatos részleteket mutatja be a dolgozat. Ez a rész röviden ismerteti a firmware két komponensének architektúráját, valamint az almodulok működését és a főbb interfészeket. Az általam tervezett almodulok a következő részben részletesebben is bemutatásra kerülnek.

A diplomaterv mérési eredmények ismertetésével zárul. A teszt két expozíció és egy képkiolvasás végrehajtásán keresztül bemutatja és összefoglalja az EDICAM rendszer működését.

Összefoglalásként elmondható, hogy a kezdeti célokat maradéktalanul sikerült teljesíteni, ugyanis a diplomatervben bemutatott implementáció megfelel a magasszintű EDICAM specifikációnak.

Abstract

This master thesis presents the details of the design and implementation of the firmware of an intelligent camera system, the so called EDICAM (Event Detection Intelligent Camera). The firmware has been implemented on two FPGAs.

The EDICAM system consists of a camera head and a development board. These hardware modules have already been available at the beginning of the development. The first part of the thesis presents some details of these hardware components as well as introduces the third party FPGA modules. These modules are responsible to handle the low level interfaces connected to the FPGAs. Firmware design on FPGAs requires specific timing considerations which are introduced as well.

The development of the firmware was based on the high level specification of EDICAM, so the second part of the thesis is focused on certain parts of this document: exposure and image readout methodology of EDICAM.

The introduction of the specification is followed by the details of the system level design. This part presents the high level architecture of the two main firmware components, and shortly introduces the submodules and the main interfaces. The submodules which are the main contributions of my development are introduced in detail in the next part one-by-one.

The last part of the thesis demonstrates and summarizes the operation of EDICAM via presenting test results: 2 exposures and a transmission of an image had been performed.

In summary, it can be said that the inital objectives have been completely achieved. The implementation introduced in this thesis complies with the high level specification of EDICAM.

Introduction

The increasing energy demand of the population is one of the most urgent problems of today. A solution is more and more important as the depletion of fossil fuel resources is expected in the close future. Fusion power plants can be considered as one of the feasible solutions to resolve the energy problem:

- the fuel of a fusion power plant is available in large amount,
- fusion plants are more environmentally friendly than nuclear plants,
- the transformation of the released energy to electricity is easy to perform.

The technology has only one huge disadvantage, namely the high technology required to realise the fusion.

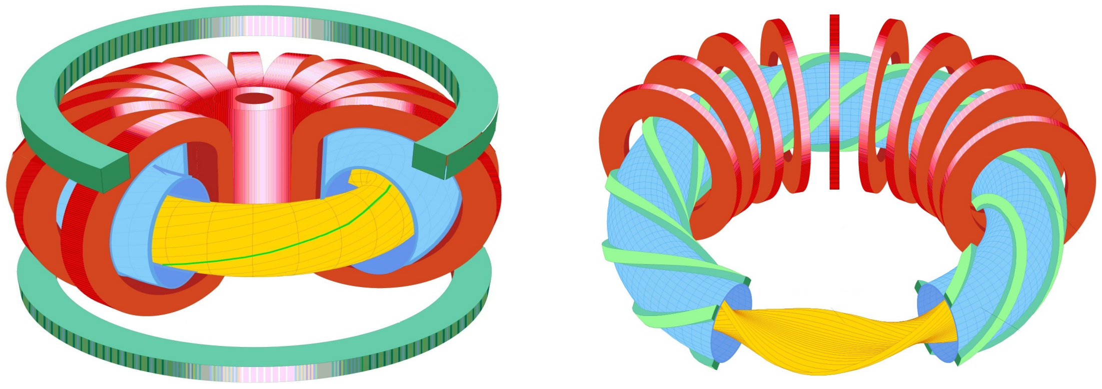

Figure 1. *Tokamak and stellarator. [9]*

The storage of hot plasma is one of the most serious problems. In figure 1. two possible solutions are exhibited. The classical tokamak is presented on left side, and the structure of a stellarator can be seen on right side. The plasma has a temperature of even 150 million degrees in these reactors [1][2].

The main purpose of the existing experimental fusion reactors is not energy production yet, but the exploration of the behavior of plasma to collect experience. The observation of such a system is a complex problem which can be performed by a camera system. The optical observation of the whole chamber of a stellarator can be performed by 10 tangential view ports. The arrangement is shown in figure 2.

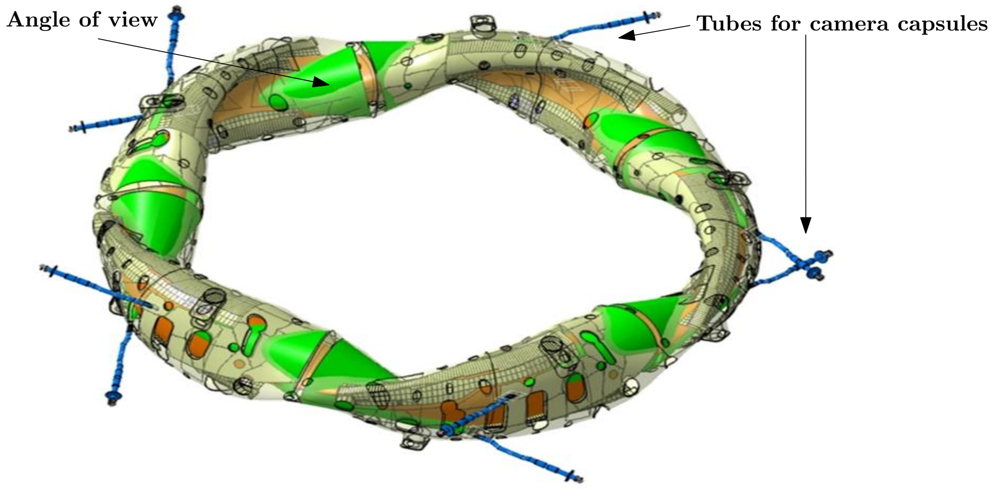

Figure 2. *Stellarator observed by cameras.*

The *MTA W FK*[1] has been started to develop a new optical observer system, the so called *EDICAM* (Event Detection Intelligent Camera). *EDICAM* is intended to be used in an experimental setup created in the Weldenstein 7-X stellarator in Greifswald. The optical sensor of the camera has a resolution of 1280*1024 pixels, and produces 12 bit intensity information per each pixel. The camera can produce pictures up to 400 times per second. Considering the numbers, approximately 3 terabytes of information is generated every hour, which has to be processed in real-time. The system is implemented on a high-end FPGA (Field Programmable Gate Array) which facilitates the realisation of low latency reactions, high bandwidth real-time communication and data processing.

I have been participated in the development of the above mentioned system at *ProDSP Ltd.*[2]. *ProDSP* has been performed the design and implementation of modules in *EDICAM* implemented on an *FPGA*. The high level functional specification along with the hardware and the low level *FPGA* modules were provided by the *MTA W FK*. This thesis presents the details of my development work which are grouped as the following:

1. Introduction of theoretical knowledge related to digital design, the applied hardware, the low level modules provided by 3rd party developers and the design flow.
2. High level specification of *EDICAM* and its interpretation.
3. System level considerations, division of the system to submodules.
4. Implementation of the submodules.
5. Test results of the integrated system.
6. Possible ways of further development.

[1]MTA W FK stands for "Magyar Tudományos Akadémia Wigner Fizikai Kutatóközpont".
[2]ProDSP Ltd. is a Hungarian R&D company, founded in 2006 as a spinoff of the Budapest University of Technology and Economics.

Chapter 1

Background

This chapter presents the basics of the development. The theoretical background of static timing analysis is presented first. The timing part is followed by the introduction of the applied hardware modules (camera head and development board). *FPGA* units (image sensor-, optical link control and *PCIe* core) provided by 3rd party developers are described as well. The applied design flow is summarized at the end of this chapter.

1.1 Synchronous sequential circuits

In this part, synchronous sequential circuits are introduced [7]. Digital circuits can be divided into combinational- and sequential circuits. By definition, a combinational circuit is a circuit whose output, after an initial transient period, is a function of the current input. In contrast, a sequential circuit has an internal state. The output of such a system is the function of the current input and the internal state as well.

Internal states can be realised by predesigned memory components or by combinational feedback loops. Modern *FPGAs* feature predesigned memory components which are carefully designed and thoroughly analysed. Combinational loops contain potential timing hazards, so they have to be avoided in synthesis.

These memory components are carefully designed and thoroughly analysed, while feedback loops contain potential timing hazards, and aren't suitable for synthesis.

1.1.1 Memory components

Basic memory components can be divided into latches and flip-flops (FFs). Since latches are not used in the firmware (they are not used in synthesis normally), the next section focuses only on *FFs*.

Functional description of D FFs

The schematic and the function table of positive-edge-triggered *D FFs* can be seen in figure 1.1. *D FFs* have a special control signal known as clock signal which is labeled as *clk* in the schematic. *D FFs* take sample of the input data when the clock signal changes from low level to high level (see function table in figure 1.1.), which is known as the rising edge of the clock. At other times, the output remains the same as the value stored in the memory at the rising edge of *clk*, so the output of a *D FF* is unchanged until the next rising edge.

There are also negative-edge-triggered *D FFs* which are controlled by the falling edge of the clock signal. The edge of clock, which causes sampling, can be referred to as the sampling edge. Since the operation of *D FFs* is controlled by the edge of signal *clk*, they are referred to as edge sensitive. Sometimes *D FFs* are referred to as *FFs* simply in the thesis.

Timing description of D FFs

In order to ensure the above detailed functionally proper operation of a *D FF*, timing considerations have to be examined. The three timing parameters, introduced in figure 1.1., are described in the following [7]:

- Clock to q delay (T_{cq}): the propagation delay required for the *d* input to show up at the *q* output after the sampling edge of the clock signal.
- Setup time (T_s): the time interval in which signal *d* must be stable before the sampling edge.
- Hold time (T_h): the time interval in which signal *d* must be stable after the sampling edge.

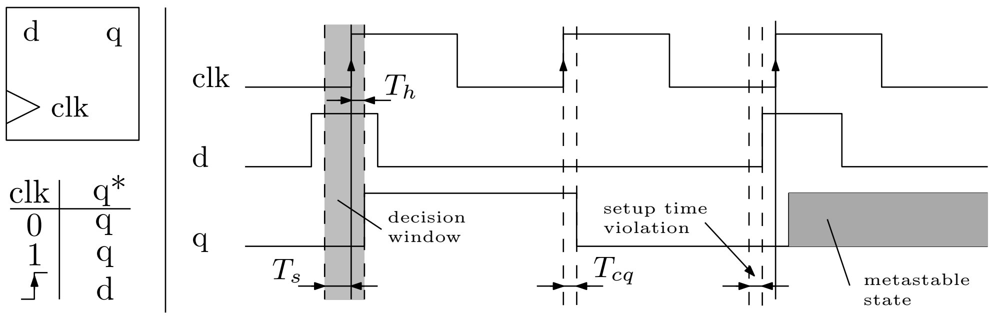

Figure 1.1. *Schematic, function table and timing diagram of a positive-edge-triggered D FF.*

A *decision window* is specified by the timing constrains (T_s and T_h), in which the input of the *D FF* has to be stable. If signal *d* changes inside this interval, the *FF* may enter a metastable state: the output is neither low nor high, which conflicts with the functional specification of the *D FF*.

In normal operation, a *D FF* is in one of the two stable states, and its output voltage is either high or low, meanwhile in metastable state, the output voltage can be anywhere in the entire voltage range. This intermediate state cannot be interpreted as either logic 0 or logic 1. If the output of the *FF* is used to drive logic, which is likely, the intermediate value may propagate to further memory elements, and lead the entire digital system into an unknown state.

1.1.2 Globally synchronous sequential circuits

A globally synchronous sequential circuit (or simply synchronous circuit), in which all *FFs* are controlled by a global synchronizing signal (the so called clock signal), greatly simplifies the design process. Synchronous design is the most suitable methodology to design complex digital systems. During the design of *EDICAM*, the principles of the synchronous sequential circuit design methodology were applied. The basics of the synchronous design methodology are summarized in this section [7].

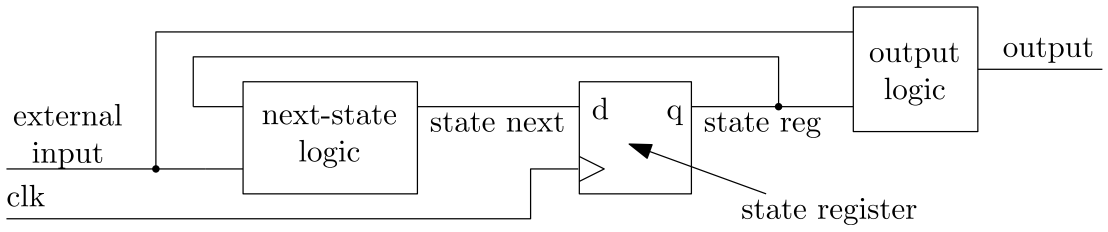

Figure 1.2. *Conceptual diagram of a synchronous sequential circuit.*

The memory element in a sequential circuit, frequently known as a *state register*, is a collection of *D FFs*. The output of the register, signal *state reg*, represents the internal state of the system. The *next-state logic* is a combinational circuit that determines the next state of the system. The *output logic* is another combinational circuit that generates the external output signal. The structure of such a system is shown in figure 1.2. During the operation of synchronous circuits, the following steps are periodically performed:

1. At the sampling edge of the clock, the value of signal *state next* is sampled and propagated to port *q*, which becomes the new value of signal *state reg*. The current state of the system is represented by the values stored in *FFs*.

2. Based on the value of signals *state reg* and *external input*, the *next-state logic* computes the value of signal *state next*, and the *output logic* computes the value of the external output.

Synchronous design is preferred because of its important advantages:

- The combinational circuits and memory elements are clearly separated. This enables the combinational part of the system to be easily isolated, and also to be designed and analyzed as a regular combinational circuit.

- Timing hazards can easily accommodated in a synchronous circuit because inputs are stored at the sampling edge of the clock. The effects of glitches are ignorable as long as they are settled in the *decision window*.

1.2 Static timing analysis

When designing a digital circuit, both the specified functional operation and the timing requirements described in subsection 1.1.1 have to be checked. The process of the examination of timing requirements is called *static timing analysis*, and can be performed by the

TimeQuest Timing Analyser. The basics and features of *TimeQuest* which were used in the design of the *EDICAM* firmware are introduced in this section [11].

The timing information necessary in a digital design can be presented in an SDC (Synopsys Design Constraints) file. This file is processed by both the synthesis and timing analyser software.

1.2.1 Introduction of timing analysis

Static timing analysis is based on the presence of clocks, namely every register launches and latches data periodically. Timing relationships between the register pairs being in connection are examined during the analysis. The principles of this method are described in the folowing, where the expressions $D\ FFs$ and registers are used equivalently.

Basic model

The sampling edge of the clock signal can be referred to as launch- or latch edge. When the transmission between two registers is examined, the sampling edge of the source register is called launch, while the sampling edge of the destination register is called latch edge. The data launched by the launch edge is intended to be sampled with the latch edge.

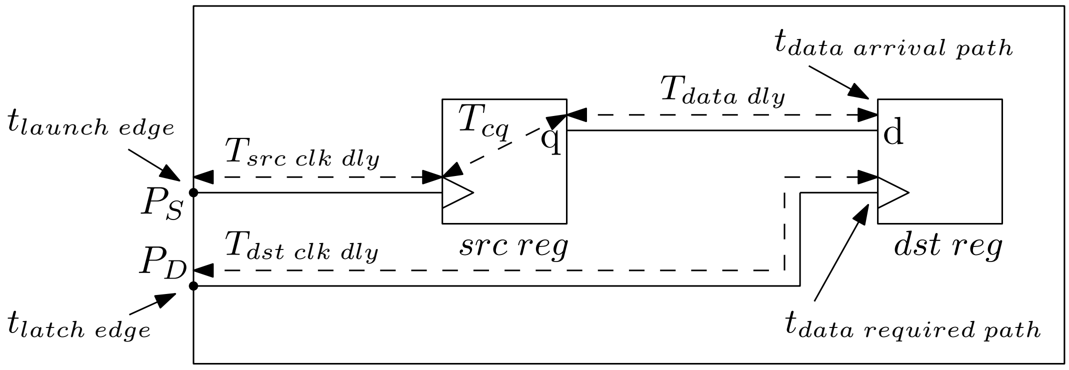

Figure 1.3. *Register pair timing model in TimeQuest.*

In figure 1.3., the model and parameters used by *TimeQuest* can be seen. Points P_S and P_D can be the same. The abbreviations $src(S)$, $dst(D)$, *dly*, and *reg* stand for source, destination, delay and register respectively. The terminology used by *TimeQuest* is not exactly the same. T_{cq}, T_s and T_h are called micro-parameters, and they are referenced as μTco, μTsu and μTh. The other parameters are called simply *launch_edge*, *latch_edge*, *src_clk_dly*, *dst_clk_dly*, *data_delay*, *data_arrival_path* and *data_required_path* regardless of the underlying meaning. Since these names used in *TimeQuest* can be confusing at first glance, the notation introduced in figure 1.3. is preferred in this thesis. The description of the tags in the figure is described in the following:

- P_S, P_D: reference points related to the timing of clocks feeding *src reg* and *dst reg*.
- *src reg*: this register launches data.
- *dst reg*: this register samples the data launched by *src reg*.

- $t_{launch\ edge}$: time of the arrival of lauch edge at P_S.
- $t_{latch\ edge}$: time of the arrival of latch edge at P_D.
- $T_{src\ clk\ dly}$: propagation delay from P_S to *src reg*.
- $T_{dst\ clk\ dly}$: propagation delay from P_D to *dst reg*.
- $T_{data\ dly}$: propagation delay from the output of *src reg* to the input of *dst reg*.
- $t_{data\ arrival\ path}$: time of the arrival of data, driven by *src reg* and triggered by the launch edge, at *dst reg*.
- $t_{data\ required\ path}$: time of the arrival of latch edge at *dst reg* (time of sample).

Timing analysis

In order to ensure the setup requirement of *dst reg*, the data launched by *src reg* has to reach the destination reg by the value of T_s earlier than the latch edge, which is described by the next inequality:

$$t_{data\ arrival\ path} < t_{data\ required\ path} - T_s \tag{1.1}$$

After substituting the parameters into (1.1), the result is:

$$\begin{aligned} &t_{launch\ edge} + T_{src\ clk\ dly} + T_{cq} + T_{data\ dly} < \\ &t_{latch\ edge} + T_{dst\ clk\ dly} - T_s \end{aligned} \tag{1.2}$$

The minimum difference between $t_{latch\ edge}$ and $t_{launch\ edge}$ is the so called setup relationship, and the deviation of the clock delays is the *clock skew* ($T_{dst\ clk\ dly} - T_{src\ clk\ dly}$).

The next data launched by *src reg* has to reach *dst reg* by the value of T_h beyond the arrival of the latch edge in order to ensure the hold requirement of *dst reg*:

$$t_{next\ data\ arrival\ path} > t_{data\ required\ path} + T_h,$$

which results in

$$\begin{aligned} &t_{next\ launch\ edge} + T_{src\ clk\ dly} + T_{cq} + T_{data\ dly} > \\ &t_{latch\ edge} + T_{dst\ clk\ dly} + T_h \end{aligned} \tag{1.3}$$

It's important to emphasize that $t_{next\ launch\ edge}$ in (1.3) represents a later time than $t_{launch\ edge}$ in (1.2). This is the consequence of the periodic operation of synchronous systems: the next launched data must not disturb the previously sent one.

The maximum difference between $t_{latch\ edge}$ and $t_{next\ launch\ edge}$ is the so called hold relationship.

Relationships detailed in (1.2) and (1.3) contain micro-parameters, delays and setup- or hold relationship. Micro-parameters are constant parameters determined by the technology, and delays can be extracted accurately from the routed design. In order to perform timing analysis, only the relationships have to be calculated by *TimeQuest* from the *SDC* file. Setup- and hold relationships can be also referred to as requirements, because the synthesis software has to satisfy the hold- and setup inequalities based on these parameters as requirements.

Writing of proper *SDC* files requires the understanding of the way *TimeQuest* calculates relationships depending on the constraints written in the *SDC*, so in the next subsections *SDC* constraints and their effects are presented.

1.2.2 Basic relationships

This subsection focuses on the derivation of the basic relationships in *TimeQuest*.

Description of clock creation

There are constraints in the *SDC* file which enable the designer to describe the waveform of the clock signal in certain points of the design. From these points, the waveform of the clock signal can be calculated anywhere in the design, because the delays of the elements in the clock paths in the routed design are known. *create_ clock*, *create_ derived_ clock* and *create_ pll_ clocks* *SDC* constraints are introduced in the following.

A schematic can be seen in figure 1.4. which represents the clock structure of a synchronous system containing a PLL[1] (Phase Locked Loop).

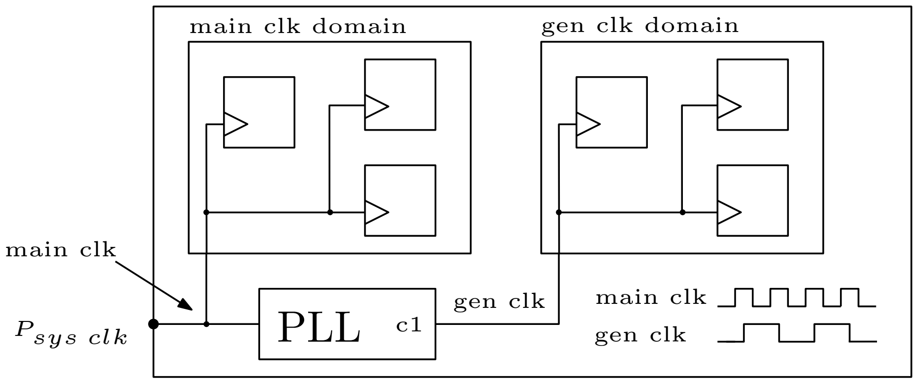

Figure 1.4. *Clock domains with a PLL.*

The clock structure of the example system is described by the following *SDC* constraints:

- `create_clock -period 8.0 -name main_clk [get_ports sys_clk]`
- `create_generated_clock -name gen_clk -divide_by 2 -phase 22.5 -source sys_clk [get_pins PLL|c1]`

create_ clock constraint is used to create base clocks, namely it can describe the waveform of the clock coming into the *FPGA*. In this particular constraint, a clock signal is created which user name is *main clk*. The target of the constraint is the *sys clk* port of the *FPGA*,

[1]A PLL is a predesigned component in FPGAs, which can be used to transform clock signals.

so at that port a clock waveform is generated with 8 ns period and first rising edge at 4 ns. The function of $P_{sys\ clk}$ is the same as the function of P_S and P_D in figure 1.3., but in this example the two points are the same.

create_ generated_ clock constraint is used to create clocks based on already defined clocks. In this example, the user name of the generated clock is *gen clk*. This clock is the modified version of *sys clk* indicated by the *-source* parameter. Because of the *source-* and *divide_by* parameters of the constraint, *gen clk* has 16 ns period, but its rising edge occurs at 1 ns caused by the *phase* parameter. The waveform is related to the same point as *main clk*, because generated clocks are analyzed as if they were coming into the device where the clock presented in the *-source* parameter. The targets of this generated clock are the registers in *gen clk domain* which are controlled by the $c1$ output of the PLL.

create_ pll_ clocks constraint describes the outputs of *PLLs*. The constraint uses parameters from the definition files of *PLLs* to determine the proper parameters of the automatically generated *create_generated_clock* constraints. These generated constraints are similar to the *create_ generated_ clock* constraint described in the previous paragraph.

Calculation of basic setup and hold relationships

Calculation of basic setup and hold relationships can be performed after the clock constraints have been applied. The steps of the derivation which are thoroughly summarized in [11] are the following:

1. The waveforms have to be drawn based on *SDC* constraints.
2. The default setup relationship is designated by the closest edge pairs where $t_{launch\ edge} < t_{latch\ edge}$.
3. The default hold relationship is designated by the closest edge pairs where $t_{latch\ edge} < t_{launch\ edge} + T_{setup\ relationship}$.

The second step is based on an inequality, but it can be done easier by the examination of waveforms. In the waveforms, an arrow has to be drawn from every launch edge towards the nearest latch edge until the time differences are started to repeat. $T_{setup\ relationship}$ is the smallest time difference between the edge pairs. The derivation of the setup relationship of the above constrained clocks is illustrated in figure 1.5.

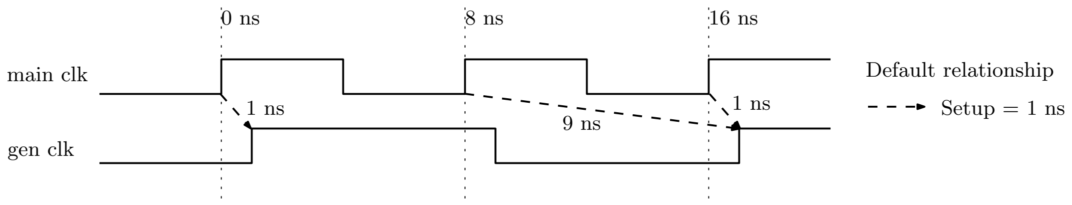

Figure 1.5. *Calculation of basic setup relationship based on waveforms.*

Not just the setup-, but also the hold requirement can be easily extracted from waveforms. The setup relationship has to be added to every launch edge first. In the second step, those latch edges have to be found which are the closest backwards in time to these shifted launch edges. At the end of the derivation, hold requirement is the difference between the closest incremented launch edge and latch edge pair designated by the previous step. This process can be seen in figure 1.6.

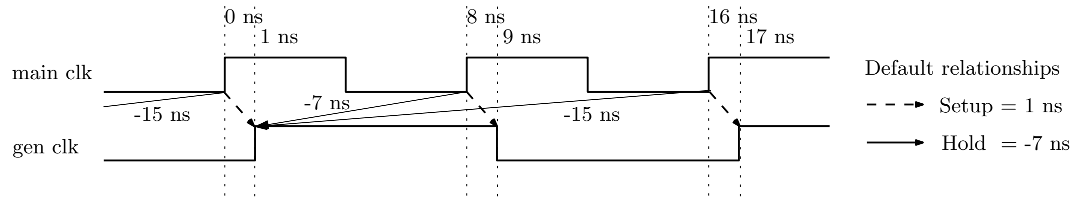

Figure 1.6. *Calculation of basic hold relationship based on waveforms.*

1.2.3 Modification of the basic relationships

In this part, some of the features of *TimeQuest* are introduced which enable the designer to modify the basic relationships.

Use of multicycle path

Step 2 of the list found in subsection 1.2.2. assumes that the designer wants to perform correct data transmissions between even the closest launch- and latch edges, namely *TimeQuest* assumes that every lauch edge can generate new output, and every latch edge samples the input. Sometimes this basic launch- latch attitude of *TimeQuest* and the derived basic relationships differ from the intentions of the designer. In these cases the default relationships can be modified by applying the mulicycle path *SDC* constraint in the design.

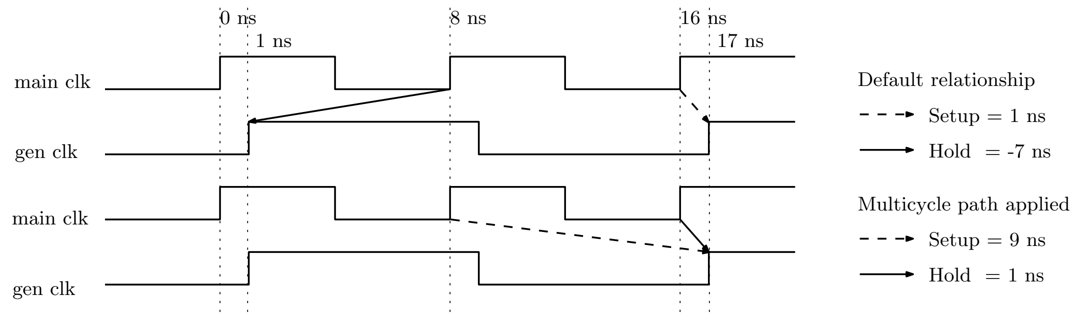

Figure 1.7. *Correction of the basic relationships.*

A clock domain structure similar to the above detailed *PLL* example (see figure 1.4.) caused problem in the *EDICAM* firmware. The solution is detailed hereafter. At top of figure 1.7., the basic relationships derived by *TimeQuest* can be seen. The default setup requirement is 1 ns which is impossible to be satisfied by the fitter, and the -7 ns hold relationship is incorrect as well. The designer has to tell to *TimeQuest*, that the next sampling edge is preferred, which can be done by the *-setup* parameter of *set_multicycle_path* constraint. This is the so called *shifting the window* method. The results of the constraint are shown at bottom of figure 1.7. The new relationships are illustrated by the arrows with common ending point, and the interpretation of these arrows is explained in the following.

The data launched by the rising edge of *main clk* at 8 ns is intended to be sampled at 17 ns by *gen clk* (the tightest setup requirement, indicated by the dashed arrow). This sampling is protected by the 1 ns hold requirement, namely the next data launched by *main clock* at 16 ns won't reach the destination register in the *decision window* around 17 ns (the tightest hold requirement, illustrated by the normal arrow), so data transmission will be performed without any time requirement violation.

Clock relationships

Basically, all clocks of the design are considered as related. If two clocks are related, *TimeQuest* will try to satisfy (1.3) and (1.2) between all registers in the two domain. If these clocks are the outputs of unsynchronised oscillators or uncompensated *PLLs*, or if the frequencies of the clocks cause impossible tight timing requirements, the clocks have to be set up as unrelated. It's important to emphasize that in these cases the designer does not want to transfer data between these domains synchronously as described in subsection 1.1.2, so the setup- and hold inequalities have to be ignored by both *TimeQuest* and synthesis software.

1.3 Applied hardware

In this section the hardware elements of *EDICAM* are introduced. The system consists of a camera head and an expanded *Altera* development board. The operation is based on high-speed links: The *Camera head* and the development board are connected via an optical link, and the development board communicates with a computer through *PCIe* interface. The system used in the design, along with its schematic representation, is shown in figure 1.9. and 1.8. respectively.

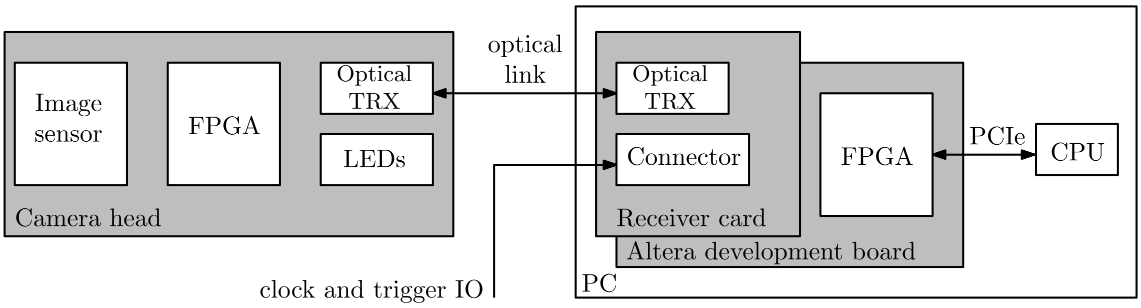

Figure 1.8. *Schematic representation of the hardware components.*

The *Camera head* contains the image sensor controlled by an *FPGA*, meanwhile the *Altera* board provides the significant portion of the computing capacity of *EDICAM*. Details of this two main modules of *EDICAM* are presented in the following subsections.

The firmware development software, the *FPGA* programmer and the debug related tools are installed on an auxiliary computer which can be seen on the left side in figure 1.9. This computer communicates with the *FPGAs* via *JTAG* and is independent from the

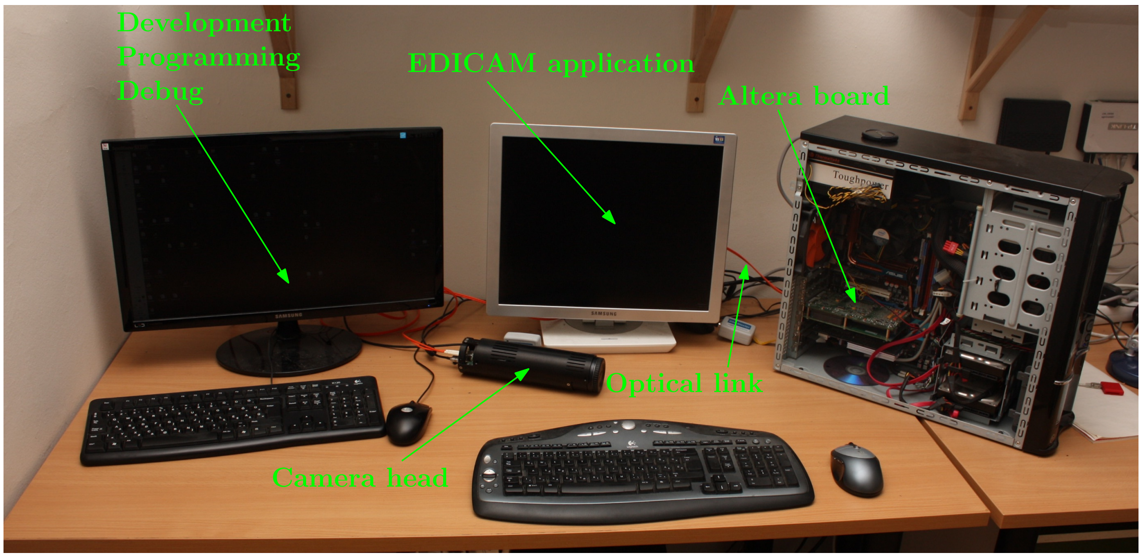

Figure 1.9. *Picture of EDICAM.*

operation of *EDICAM*. The computer connected to the *EDICAM* via *PCIe* can be seen on the right side.

1.3.1 Camera head

This subsection introduces the main components of the *Camera head*. The description is completed with the simplified block diagram of the *Camera head* (see figure 1.10.)

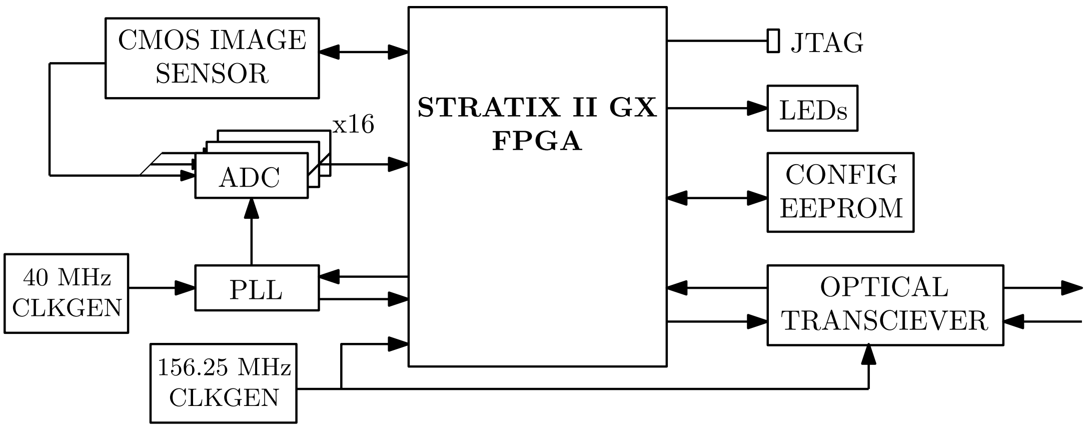

Figure 1.10. *Simplified block diagram of the Camera head.*

The image sensor is a *LUPA*1300 1.3 megapixel high speed CMOS device (see figure 1.11.) which is based on a 1280 by 1024 pixel array [8]. The sensor can produce images up to 450 times per second at full resolution. The photodiodes can be in two different states, collecting light or reset. The current value of the diode array, determined by the state of exposure[2], can be loaded to a storage by sampling. The captured analog values are valid only for approximately 2 ms. The content of the storage is transmitted via 16 analog parallel output amplifiers which are digitized by the 12-bit ADCs (Analog Digital Convert-

[2]Exposure is the process when light is collected in the light sensitive detector array.

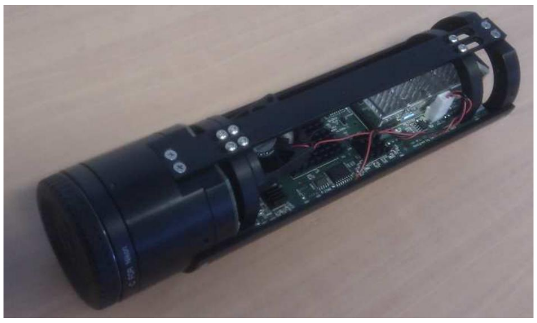
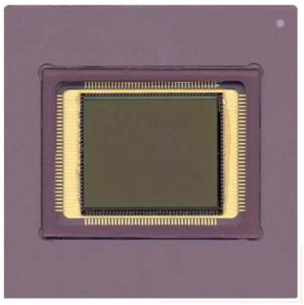
Figure 1.11. *Camera head and image sensor.*

ers). The amplifiers have a pixel rate of 40 MHz. The *LUPA*1300 can produce the image data of rectangular subareas as well, which enables the sensor to increase the achievable sampling frequency. Such an area is defined by a so called ROI (Region Of Interest)[3].

The transmission of the image data requires a very fast interface between the *Camera head* and the Altera board which is realised by an optical transciever which support speeds up to 10 Gbps. The image sensor, optical transciever and the other modules in the *Camera head* are controlled by an *EP2SGX30 FPGA* which is related to the *Stratix II GX FPGA* family. *Stratix II GX FPGAs* are introduced in subsecion 1.3.3. The logical capacity enables the *FPGA* in the *Camera head* to implement some basic parts of the functionality of *EDICAM* next to the necessary handling of the image sensor and the optical link.

The *FPGA* can be configured directly via *JTAG* or it can load the configuration file from the config *EEPROM* independently.

1.3.2 Expanded Altera development board

This part focuses on the introduction of the expanded *Altera* development board [5]. The picture of the board is shown in figure 1.12. which contains tags only for units applied in *EDICAM*. The optical interface of the board is not used, because it cannot provide the transmission speed required by the specification. A faster optical connector (10 Gbps) along with the trigger and clock ports are accommodated in the extension card (*Receiver card* in figure 1.8.). This is a custom hardware component which is connected to the *Altera* board.

The *Altera* board contains a 512 Mbyte flash which can store the configuration files of up to 8 designs. The actual configuration can be selected via the configuration switch which is located at the back plane. The autonomous configuration is supported by a *MAX II* device. This unit is used exclusively to configure the flash or the *FPGA*. The latter is performed very quickly, because the *FPGA* has to be ready within 80 ms beyond power-on reset (according to the *PCIe* specification). The communication towards the *PC* is supported by a *PCIe* interface with 8 lane. This port facilitates the complete transmission

[3]For clarity, it has to be emphasized, that the expression *ROI* has different meaning for the *LUPA*1300 and *EDICAM*. As described above, *LUPA*1300 can manage only rectangular *ROI*s, while a *ROI* in *EDICAM* can have arbitrary shape. Where it is unequivocal, hints are presented to aid the clear understanding.

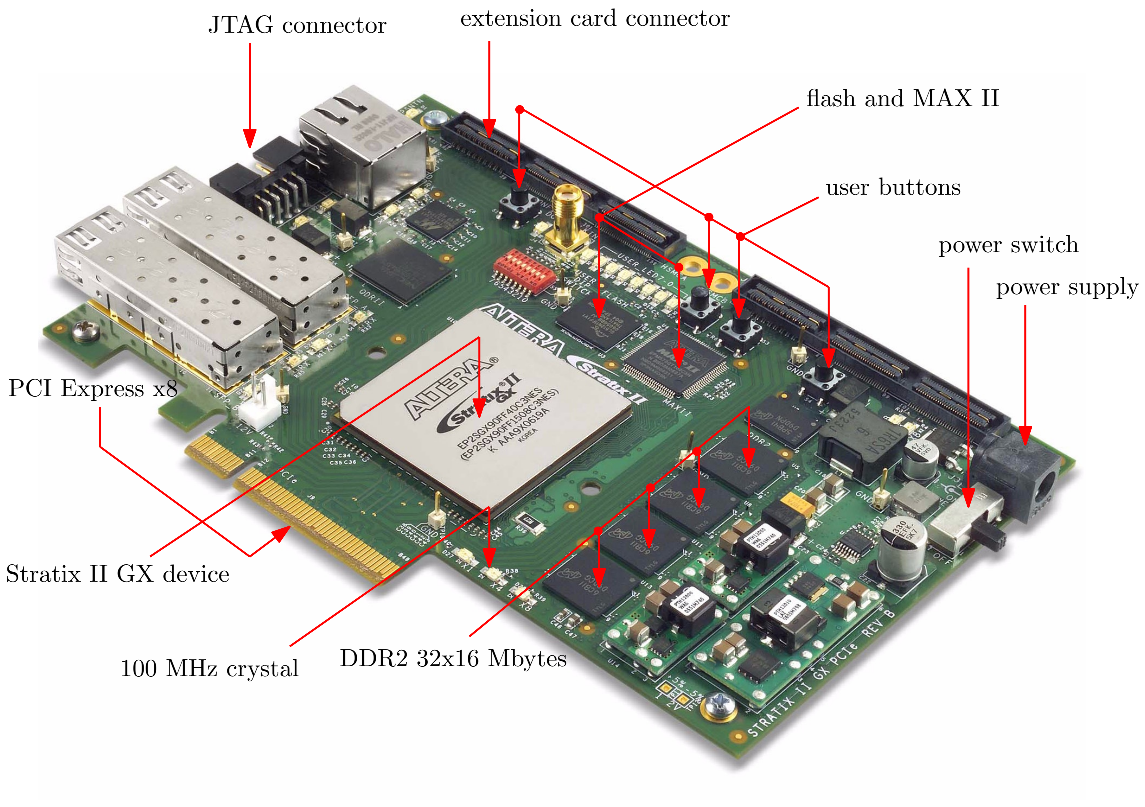

Figure 1.12. *Altera development board. [5]*

of the image data. The reference clock of the *PCIe* interface is provided by an oscillator with a frequency of 100 MHz which can be routed to the gigabit transcievers. The other clock source with a frequency of 156.25 MHz is utilized by the low level optical interface.

The board features a *JTAG* chain, which supports in system debug and programming: Altera provides powerful debug tools which utilizes this serial interface. The *MAX II CPLD*[4] and *FPGA* are serially located in the chain.

The memory capacity of the *FPGA* is expanded with 256 Mbyte DDR2 memory which supports low latency, high bandwidth storage of high amount of information. This capacity consists of 72-bit synchronous dynamic random access memory (SDRAM) which is made up of four 16-bit interfaces. The maximum clock frequency is 333 MHz, which means a theoretical bandwidth of 48 Gbps.

The core of the Altera board is an *EP2SGX90 Stratix II GX FPGA*. This module gives the main part of the computing capacity of *EDICAM*, so the functionality (e.g. data processing) is implemented on this module mostly.

1.3.3 Introduction of Stratix II GX FPGA family

Stratix FPGAs are the high-end *FPGAs* provided by *Altera*. With the *Stratix II FPGA*, *Altera* introduced a new logic cell architecture, namely the adaptive logic module. *Altera* extended the capabilities of these *FPGAs* with on-chip transcievers creating the

[4]Complex programmable logic devices (CPLDs) are less complex than *FPGAs* but have non-volatile configuration memory.

Stratix II GX FPGA. Since *EDICAM* includes high speed optical and *PCIe* interfaces, this extended *FPGA* family is applied in the system. The main features of these devices are described hereafter [4][12].

The introduced properties are memory capacity, clocking features, arithmetic support and *IO* capabilities. The dedicated storage elements are the so called *TriMatrix* memories which supports three *RAM* block sizes. These modules enable the implementation of true dual-port memories and *FIFOs* up to 550 MHz. The *Sratix II GX* family provides up to 16 global clock in all, and up to 32 regional clock networks per device region. The high-speed *DSP* blocks operate up to 450 MHz. These elements can accommodate e.g. multiply-accumulate functions or *FIR* filters. The superior *IO* blocks support even high-speed external memory interfaces, such as quad data rate *SRAM*, double data rate *SDRAM* and single data rate *SDRAM*. High speed serial interfaces are supported by up to 20 high-speed serial transciever channels which provide up to 255 Gbps of serial bandwidth (6.375 Gbps in each direction per each transciever). The introduced *FPGA* family supports multiple intellectual property megafunctions from *Altera MegaCore* functions which simplifies the design process.

High level architecture

Stratix II GX devices have a two-dimensional row- and column based architecture as can be seen in figure 1.13.

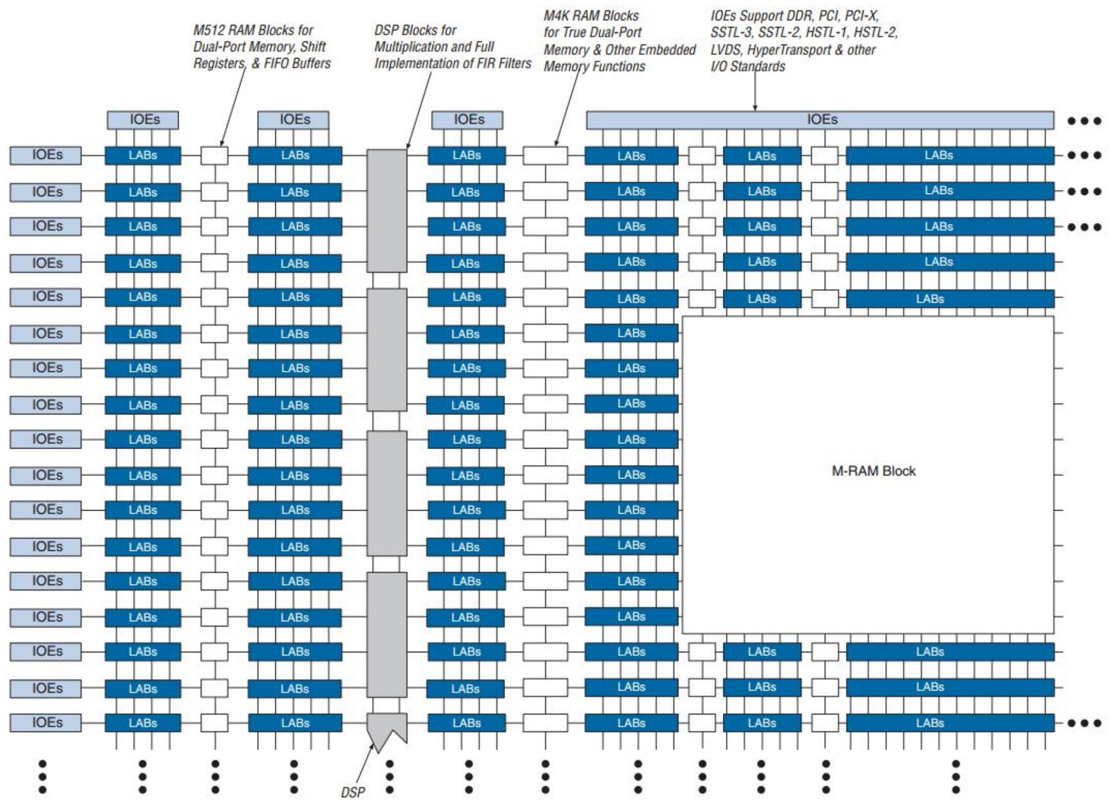
M512 RAM Blocks for Dual-Port Memory, Shift Registers, & FIFO Buffers
DSP Blocks for Multiplication and Full Implementation of FIR Filters
M4K RAM Blocks for True Dual-Port Memory & Other Embedded Memory Functions
IOEs Support DDR, PCI, PCI-X, SSTL-3, SSTL-2, HSTL-1, HSTL-2, LVDS, HyperTransport & other I/O Standards
IOEs
LABs
M-RAM Block
DSP

Figure 1.13. *Block diagram of Stratix II GX architecture. [4]*

The basic building elements are the logic array blocks (LABs), memory blocks ($M512$, $M4K$ and M $RAMs$) and DSP blocks. The DSP blocks can implement four 18 bit x 18 bit or one 36 bit x 36 bit multiplier. Each LAB consists of 8 adaptive logic modules (ALMs) which is the basic building block of logic.

The memory blocks have a capacity of 0.5 K, 4 K and 512 K without parity bits (K equals to 1024 bit). *M4K RAM* and *M-RAM* blocks provide memory interfaces up to 60-bit and 144-bit wide respectively, and they support all of true-dual port, simple dual port and single port modes. In contrast, $M512$ can not implement true-dual port mode. The memory blocks except *M-RAM*, DSP modules and $LABs$ are placed in a column structure, while *M-RAM* is located individually in the logic array of the device.

The IO pins of the $FPGA$ are fed by IO elements (IOEs) located at the end of LAB rows and columns around the periphery of the device. Various single-ended and differential IO standards are supported. Each IOE consist of a bidirectional IO buffer, six registers for registering input, output and output enable siganls. These registers are controlled by a dedicated clock, and they allow the realisation of high performance external memory interfaces such as DDR, $DDR2$, *QDR II SRAM*.

The resources available in the particular devices of $EDICAM$ are listed in table 1.1. It is obvious, that the $GX90$ located in the *Altera* board is much powerful than the $GX30$ in the *Camera head*. The difference between the two devices has a factor of almost 3 in every listed resource. The functionality of $EDICAM$ is mostly realized by the capacity of the $GX90$. It has to be added that the $PCIe$ soft core and the related application layer with DMA in the $GX90$ consumes almost 30 % of the resources. Furthermore, this part operates at high frequency (250 MHz). In summary, the extra resources are consumed by the $PCIe$ interface and the implementation of $EDICAM$ functionality together.

Device	**M512**	**M4K**	**M-RAM**	**MUL 18x18**	**ALM**	**Memory capacity**	**Transcievers**
EP2SGX30	202	144	1	64	13552	1.47 Mb	4/8
EP2SGX90	488	408	4	192	36384	4.52 Mb	8/12

Table 1.1. *Summary of resources in the applied SIIGX FPGAs.*

It is important to localize these devices in the world of $FPGAs$. To illustrate the relative capacity of $EP2SGX30$ and $EP2SGX90$, the ordinary *Spartan* $3E$ 250 and the most advanced *Altera* $FPGA$ is compared with the $GX90$ in table 1.2.

Device	**MUL 18x18**	**LEs**	**memory**	**PLL**
XC3S250E	12	5508	220 kb	4
EP2SGX90	192	90960	4.52 Mb	8
5SGXBB	704	952000	53 Mb	32

Table 1.2. *Resources of FPGAs with different level.*

It has to be emphasized that the comparison of these $FPGAs$ via the numbers listed in the table is inaccurate[5]. These $FPGAs$ have very different architectures (logic element

[5]Equivalent logic elements (LEs) is the logic capacity expressed in a four-input LUT-based architecture

structure, memory block structure, etc.) and technology (operating frequency, power consumption). These aspects not included in the table further increase the difference. Considering only the numbers, it can be seen, that there are *FPGAs* which are bigger or smaller relative to *GX*90 with an order of magnitude. It is remarkable that the performance of even these *Stratix II FPGAs* is negligible compared to 5*SGXBB*.

Introduction of LABs and ALMs

When designing with *FPGAs*, the understanding of the underlying structure is necessary to reach proper implementation. The description provided in the following is only an introduction.

The main logic building block of *Stratix II FPGA* is introduced in detail in the following [3]. Each *LAB* consists of 8 *ALMs*, a carry chain, a shared arithmetic chain, *LAB* control signals, local interconnects and register chain connection lines. As illustrated in figure 1.14., each *ALM* contains a variety of *LUT* based resources, two registers, two dedicated full adders, a carry chain, a shared arithmetic chain, a register chain and additional logic units. These resources can be adaptively divided into two adaptive *LUTs* (ALUTs). The *LUT* based resources and the additional logic units are tagged as *Combinational logic* in the figure.

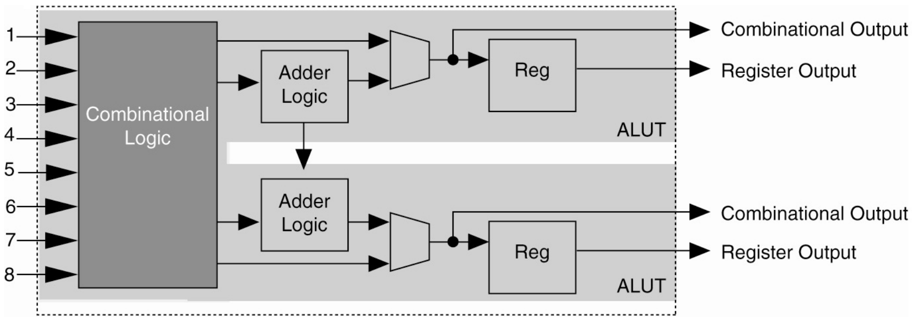

Figure 1.14. *Relationship of ALMs and ALUTs. [3]*

The structure of *LABs* allows the Quartus II Compiler to implement arithmetic functions and shift registers and certain functions very effectively. This is obtained by the placement of associated logic in the same or adjacent *LABs* which enables the common use of the above enumerated chains. The *GX* family further increases the logic utilization by the so called register packing: Every *ALM* in the device can use both its *LUT* and register resources for independent functions. This is beneficial when the original function uses only the combinational part of the *ALM* and hence the corresponding register would be wasted.

The power of *ALMs* lies in the adaptive *Combinational logic* which enables the structure to implement various combinations of two functions with up to 8 inputs. The adaptive logic is always divided by the compiler in a manner which minimizes the amount of unused resources. Some of the possible results of *ALM* configurations are the following:

(like the structure in Spartan 3). *LUTs* are embedded asynchronous memory elements which are used to implement logic functions.

- Two independent 4-input *LUT* (4-*LUT*) which ensures the backward compatibility.
- An independent 3-*LUT* and 5-*LUT*.
- 5-*LUT* and 4-*LUT* with one common input.
- Two 5-*LUT* with two common inputs.
- Any six-input functions.
- A subset of seven-input functions.

1.4 Reused FPGA modules

This section introduces the modules of the firmware which are provided by 3rd party developers (*SCFW*, *10G link*, *PCIe core*). The reused *FPGA* modules are illustrated in figure 1.15. (dashed boxes). The next descriptions focus on the interfaces and the high level functionality of the modules. Most of the used documentations related to this section are confidental and can not be referenced unless it is separately indicated (e.g. *PCIe* related modules provided by Altera).

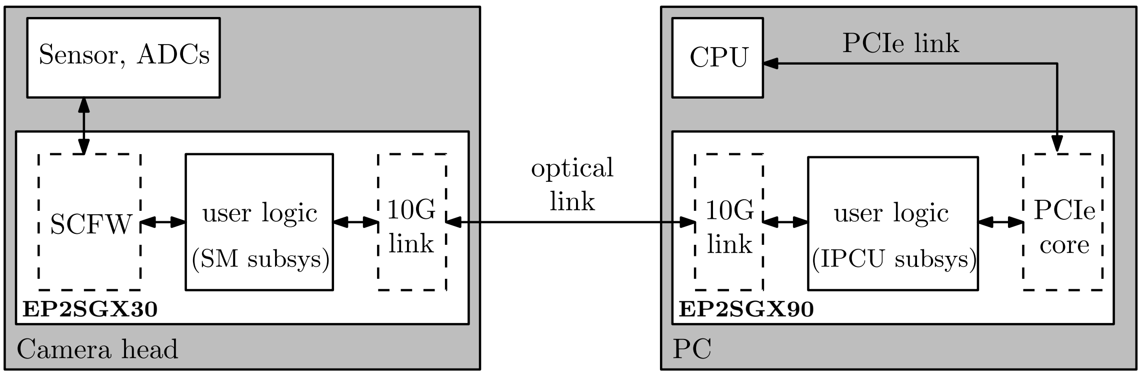

Figure 1.15. *Reused FPGA modules in EDICAM.*

1.4.1 10G link

The *10G link* provides a high level interface between the optical link and the firmware which covers the handling of the low level *XGMII* interface of the optical link. These interfaces are illustrated in figure 1.16.

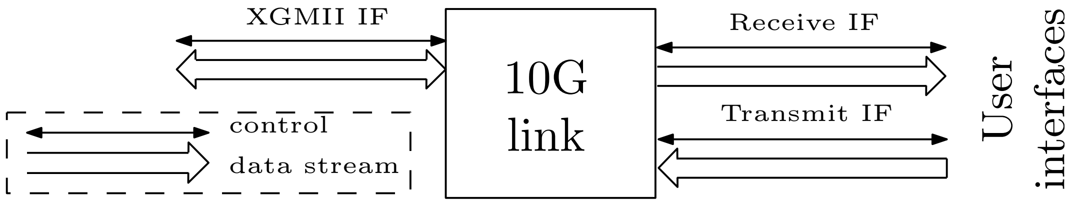

Figure 1.16. *Interfaces of 10G link.*

Towards the firmware, only simple high speed streaming interfaces are provided in both receive and transmit directions. The streaming interfaces have a maximal speed of 10 Gbps, which enables *EDICAM* to produce high resolution images up to 400 times per second.

The streaming ports are implemented as 64-bit *FIFO* interfaces with 2 auxiliary bits which can be used to transmit data packets easily. The units connected to 10*G link* have to match the interface described in the following.

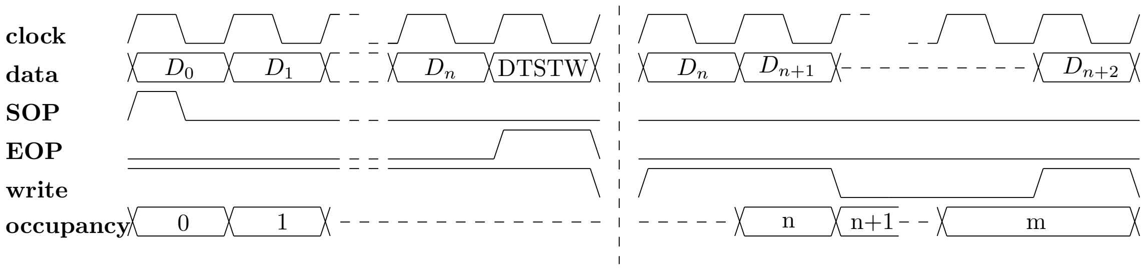

Figure 1.17. *Waveform of a write transaction on* 10*G link.*

A transmission of a packet is illustrated in figure 1.17. from the transmitter point of view. In the first cycle, *SOP* (start of packet) is asserted which marks the first QW [6] of the data packet. If *write enable* is kept high, the value of the data input is written into the link. The transfer can be suspended any time by deasserting *write enable. occupancy* has to be monitored continuously: when the high threshold level is reached, *write enable* has to be deasserted until the first cycle after the occupancy level drops below the low threshold level (cycle 2 and n in figure 1.17.). These threshold levels are constants provided by the supplier of *10G link*. At the end of the transaction, *EOP* (end of packet) has to be asserted when writing the last *QW*.

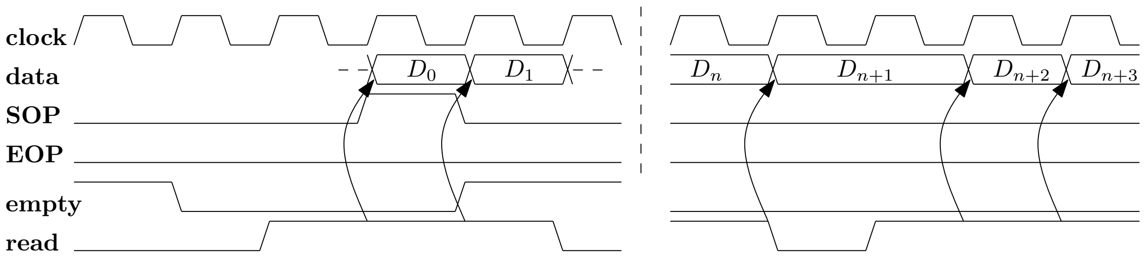

Figure 1.18. *Waveform of a read transaction on 10G link.*

If a packet is received via the optical link, the *QWs* can be accessed via the receive streaming interface. An example waveform of a receive transmission is shown in figure 1.18. The reader has to monitor the *empty* signal which indicates whether a new *QW* is available in the *FIFO*. The read operation can be started immediately after *empty* transitions to low (cycle 3). *read* can be asserted continuously until *empty* is asserted again (cycle 5). The read process can be suspended at any time by deasserting *read* (cycle 2-3). Along with the data *QWs*, *SOP* and *EOP* bits are also provided in the same positions as in the transmitter before the transmission.

[6] QW stands for quadword which is a group of 64 bits.

1.4.2 Sensor Control Firmware (SCFW)

The low level control of the image sensor and the corresponding *ADCs* is performed by the *Sensor Control Firmware*. The high level interface provided by this module enables *user logic*, without knowing the underlying details, to perform image data read with ease.

In figure 1.19., the user interfaces are tagged, meanwhile the control signals of the sensor and *ADCs* are omitted. Exposition and sampling are controlled via *Exposure and sample control interface*. The *ROI* to be read can be defined via *ROI parameter FIFO interface* after a sampling occurs. The number of *ROIs* which are read between two samplings is arbitrary. After the *ROI* parameters have been sent, the pixel data extracted from the *ADCs* are accessible on *Image data FIFO interface*. The state of the analog storage, located in the sensor, is tracked by *SCFW*, so the pixels which are received too late are marked. This subsection describes these user interfaces in detail.

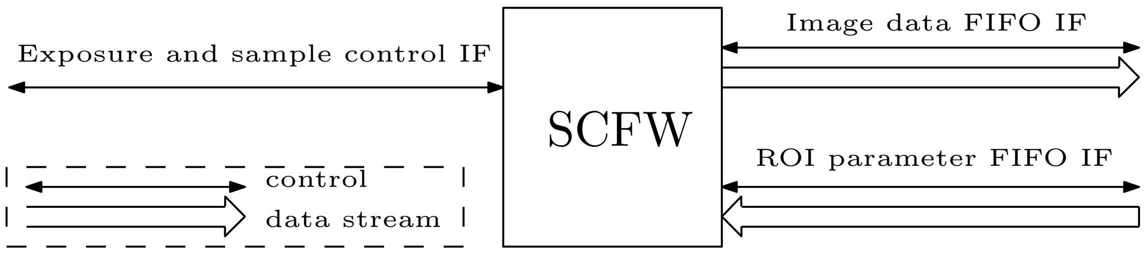

Figure 1.19. *User interfaces of SCFW.*

Exposure and sample control interface consists of *busy_o*, *sample_i* and *exposure_i* signals. The signal names are postfixed with "_i" or "_o" which indicates the direction of the signals from the *SCFW* point of view. The light sensitive diode array collects the light during *exposure_i* is asserted. Since the photodiodes are reset while *exposure_i* is deasserted, exposure cannot be paused. A sample can be taken any time by an impulse on *sample_i* which copies the content of the array to an analog storage in the sensor. Sampling and exposure can be performed independently from the state of the sensor, *SCFW* and other signals, so the appropriate control of these signals is the responsibility of *user logic*. Sampling should not be performed while *busy_o* is kept high, which is explained later.

user logic can define rectangular areas, the so called *ROIs*, on *ROI parameter FIFO interface*. The transmitted parameters consist of the horizontal and vertial coordinates of the upper left corner and the extension of the area to be read. In the case of *ROIs* with arbitrary shape[7] the marking of the first and last subareas is supported by framing signals. These signals are auxiliary bits of the data port of the *parameter FIFO*. The corresponding pixel data segments via *Image data FIFO interface* will be marked required by these framing signals.

SCFW monitors continuously the state of the *ROI* parameter *FIFO*, and if a new descriptor is available, it will be processed immediately. This task consists of the following steps: The descriptor is downloaded to the *LUPA1300* first. After this step has been finished, pixel data is clocked out both from the sensor and *ADCs*. *busy_o* is kepth high

[7] *EDICAM* manages *ROIs* with arbitrary shape (*EDICAM ROI*), however *SCFW* supports directly only the readout of rectangular areas (*SCFW ROI*).

during the whole procedure, and is deasserted when the *ROI* is completely received from the sensor. Since the source of the image is the analog storage in the sensor which is overwritten by sampling, the assertion of *sample_i* control signal during *busy_o* is active corrupts the reading of the current *ROI*.

The information transmitted by the sensor are buffered in *SCFW*, and are accessible via *Image data FIFO interface*. The pixel intensities are presented in 192-bit segments which accommodate data of 16 pixels with a resolution of 12-bit per each pixel.

The data port is extended with framing bits as mentioned at *parameter interface*. The start bit is asserted when the first image data segment is performed related to a particular *SCFW ROI* which has been defined with the start bit kept high on the *parameter interface*. The same applies to the end bit, except it can indicate the last image data segment of a *ROI*.

SCFW tracks the state of the sample in the analog storage as previously mentioned. An extra bit of the *Image data* port indicates whether the current pixel bits were generated beyond the period of validity.

SCFW has lower level control and status signals as well, namely two reset and a ready signals. The introduced user ports of the module can be used during the ready signal is kept high, and the operation can be restarted by the reset inputs.

1.4.3 PCIe core

The communication between *IPCU subsys* and the *CPU* is supported by a high speed *PCIe* link (see figure 1.15.). This subsection shortly introduces the main characteristics of this subject in the following.

The complete implementation of a module which complies the *PCIe* specification is itself an enormous task. Vendors provide soft- and hard *PCIe* cores[8] which implement certain layers of the protocol to increase productivity.

The *Altera* development board is supplied with the *Altera PCIe* compiler which allows the designer to create and customize *soft PCIe* cores in a simple manner. The generated *PCIe core* in *EDICAM* implements both the required and optional features of the *transaction*, *data link* and *physical* layers. The top level *transaction* layer is completed with an example application layer which further simplifies the design [6].

The interfaces of the *PCIe* core are summarized in figure 1.20. and introduced hereafter.

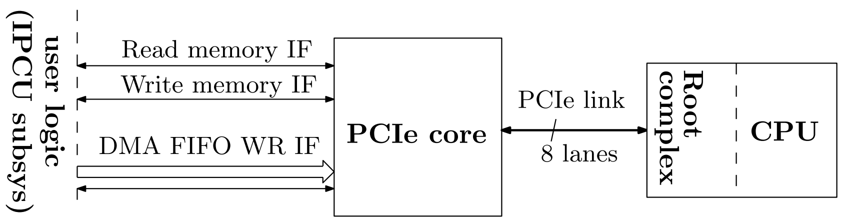

Figure 1.20. *Interfaces provided by the PCIe core.*

[8]Hard cores are prefabricated dedicated submodules on the silicon, while soft cores are configurable *HDL* modules which has to be implemented by general *FPGA* resources. Hard *PCIe* cores are featured only by the superior *FPGAs*.

The example application handles both simple memory and complex *DMA*[9] access. These features are used by internal submodules in the original application, and hence modifications had been performed. A memory- and a *FIFO* interface is provided in each direction in the customized module, which supports both fast data transfers and simple reads and writes between the application running on the *CPU* and *EDICAM*.

1.5 Design flow

FPGA development consists of various stages. This section summarizes the steps of the applied design flow which can be seen in figure 1.21. [10].

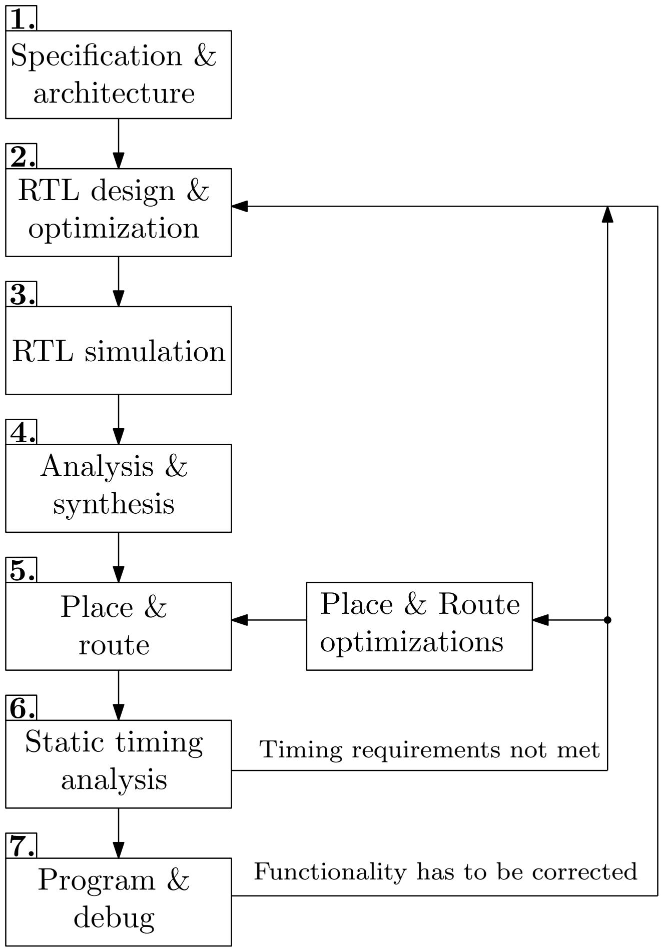
1.
Specification & architecture
2.
RTL design & optimization
3.
RTL simulation
4.
Analysis & synthesis
5.
Place & route
Place & Route optimizations
6.
Static timing analysis
Timing requirements not met
7.
Program & debug
Functionality has to be corrected

Figure 1.21. *FPGA design flow.*

1. The development of *EDICAM* was started with the interpretation of the *high level specification* (provided by *MTA W FK*). It is important to see the system in the same way as the customer. The architecture has to be clearly defined before the *RTL*[10] design can be started, so the partitioning of the firmware into submodules have been performed in the beginning of the flow as well.

[9]Direct memory access (DMA) in the *PCIe* protocol is the opportunity to launch independent memory write packets toward the root complex.

[10]Register transfer level (RTL) is a design abstraction which models a synchronous circuit in terms of registers and logic operations performed on the outputs of the registers.

2. The functionality of each submodule has to be implemented in *VHDL* hardware description language. The code intended to run on an *FPGA* has to be synthesizable. This means that conscious coding stlye has to be applied, which fundamentally influences the performance of the resulting system.

3. The functionality of the synthesizable modules has to be verified before the *FPGA* implementation. The reliable operation of the system requires the thorough examination of each module. This step of the flow is completed by system level simulations which ensure the appropiate operation of the integrated system. Since the compilation time of such a large design is significant (around an hour), iterations towards the end of the design process is much more expensive than in this early stage. This causes the importance of simulations. The simulations were performed by *ModelSim*.

4. Syntax and semantic errors are checked at the beginning of this phase, and the design hierarchy is identified. The *VHDL* code is translated to different types of *RTL* primitives, including counters, adders, block *RAMs*, multipliers, state machines and registers. Technology mapping is carried out on these *RTL* primitives which implement the design using device specific resources, such as *ALMs*, *LEs* (*ALUT* with a register) and other dedicated logic blocks. This step was performed by the integrated synthesizer of *Quartus II*.

5. The design specific resources have to be mapped to the actual device. The fitter assigns physical locations to these resources and connects them appropriately via routing. Timing constraints are considered during the operation of fitter which practically controls the fitting process (actually the synthesizer is affected as well). This step was performed by the integrated fitter of *Quartus II*.

6. Compliance of the routed design with the timing constraints has to be checked before programming the *FPGA*. This step was performed by *TimeQuest Timing Analyzer*.

7. The monitoring and control of certain signals of the *FPGA* design is necessary to validate and debug the operation. The observation of the programmed device was supported by *SignalTap II Logic Analyzer*, and control functions were provided by *SignalProbe*.

The introduced design flow contains feedbacks, which is shown in figure 1.21. Further iterations have to be performed when *Step* 6. reports that the timing requirements have not been met by the routed design. This problem can be resolved by modifying the options of the *Quartus II* fitter, e.g. increasing the placement effort. If the fitter is unable to place and route the design appropriately, the *RTL* design has to be further optimized. Both *RTL* level modifications and increased fitter efforts can be very time consuming, so the importance of the careful design in the early stages of the flow has to be emphasized again. Further *VHDL* code modifications can be made necessary by bugs as well which have been came to light only after programming.

Chapter 2

High level EDICAM specification

EDICAM performs periodic image readout and also calculates some basic parameters of the information read. The operation is controlled by external control signals, parameters set up by the *PC*, and by the results of the calculations which are performed in real-time. In this chapter, the high level specification of the firmware developed by *ProDSP Ltd.* is introduced.

The specification of *EDICAM* is provided by *MTA W FK*. These are confidental documents which can not be referenced. The subsections of this chapter (except 2.1) are the summary of those parts of the original specification which are relevant to my work in the *EDICAM* project.

2.1 Introduction of the main modules of the EDICAM firmware

EDICAM consists of two main components, therefore the firmware is also divided into two main modules: *Sensor Module* (SM) implemented on the *Camera head* and *Image Processing and Control Unit* (IPCU) realised on the *Altera* development board. The relationships between the firmware- and hardware modules are illustrated in a high level block diagram which can be seen in figure 2.1. The two systems operate in a master-slave manner, because the commands initiated by *IPCU* are performed by *SM*.

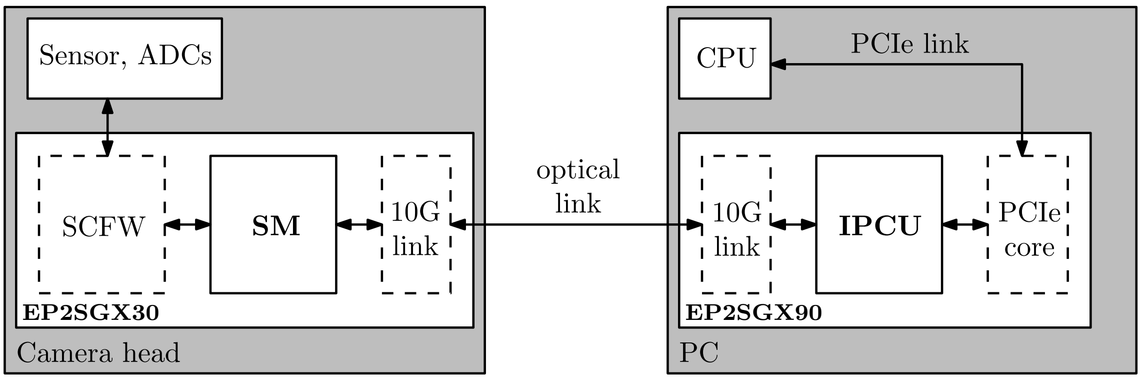

Figure 2.1. *Function of the developed firmware modules in EDICAM.*

Basically, *IPCU* controls exposure, sampling and readout[1] performed in the *SM*. The image received can be processed and sent to the *PC*. Readouts are controlled by processes running parallel which can have different parameters (image shape, readout period, etc). These and other parameters have to be set by the *PC* via a register table. The operation of the system is also controlled by external inputs and results of image processing.

SM provides a high level interface between *IPCU* and *SCFW*. This interface enables *IPCU* to control the sensor by relatively high level parameters packed into commands. These commands are decoded by the *SM*, then exposure, sample and readout is controlled as required by the parameters.

2.2 Common parts

This section introduces parts of the specification which are related to both *IPCU* and *SM*. Details related to timing in *EDICAM* are presented.

2.2.1 Timing

Both *SM* and *IPCU* have an internal 64-bit counter (the so called *ETU* (*EDICAM* Timing Unit) *counter*) with 100 ns resolution. This period is the unit of the system time in *EDICAM*, and is referred to as *ETU*. *ETU time* (*EDICAM* system time) is the value of the *ETU counter*.

These counters can be reset by an external input signal or by software command. The counters of the two modules are synchronized through the 10*G link*. Timing of exposure and data readout is based on these counters.

The *ETU counters* can be clocked by either an oscillator on the development board or an external clock input received by the extension card. This source applies only for the time measurement, and does not affect the clocking of the *FPGA*. The source clock drives a *PLL* which ensures the proper 100 ns resolution (10 MHz) of system time even with source clock periods not equal to 100 ns.

2.3 IPCU related parts

Parts of the specification which are related mainly to *IPCU* are presented in this section. The description focuses on the details of image generation (*readout*) and real-time processing (*events*).

2.3.1 ROIP concept description

The core feature of *EDICAM* is the execution of the so called *Region Of Interest Processes* (ROIPs). These tasks enable the *PC* to control the *readouts* via a high level interface. This subsection presents the details of the *ROIP* concept. The development of the modules implementing the functionality of *ROIPs* is out of the scope of my work. This subsection summarizes both the specification of *ROIPs* and the relevant characteristics

[1]A readout involves the entire transmission of parameters and image data related to a ROI.

of the corresponding implementation. The details introduced are necessary to comprehend the operation of other modules, especially in the case of parts in this chapter which describe the *readout* operation in the *SM*.

ROIPs are generating *readout requests* one-by-one which are eventually executed by the *SM* as a *readout*. A *readout request* contains all the informations necessary to perform a *readout* in the *SM*. The time, when a *readout request* is sent is the *launch time*, while the actual time of the *readout* is referred to as *sampling time*.

The *PC* has to specify the area to be read and the timing of the *readout* generation. The timing parameters set by the *PC* are t_{start}, T_{period}, N_{loop}, *immediate* and *normal* attributes, while $T_{preload}$ is a constant. These parameters are introduced later.

The *defined* flag of a *ROIP* can be set after all of the parameters introduced hereafter are configured. It has to be emphasized that a *ROI* in the scope of *SM* and *IPCU* (*EDICAM ROI*) can have more complex shapes than rectangular. If a *ROI* has arbitrary shape (unlike simple rectangular *ROIs* introduced in *SCFW*), each row of the area is defined as a rectangle with one pixel height in the description (*SCFW ROI*). The relationship between the various types of *ROIs*, full image and *sub-ROI* is shown in figure 2.2. *Sub-ROIs* are introduced later in subsection 2.3.2.

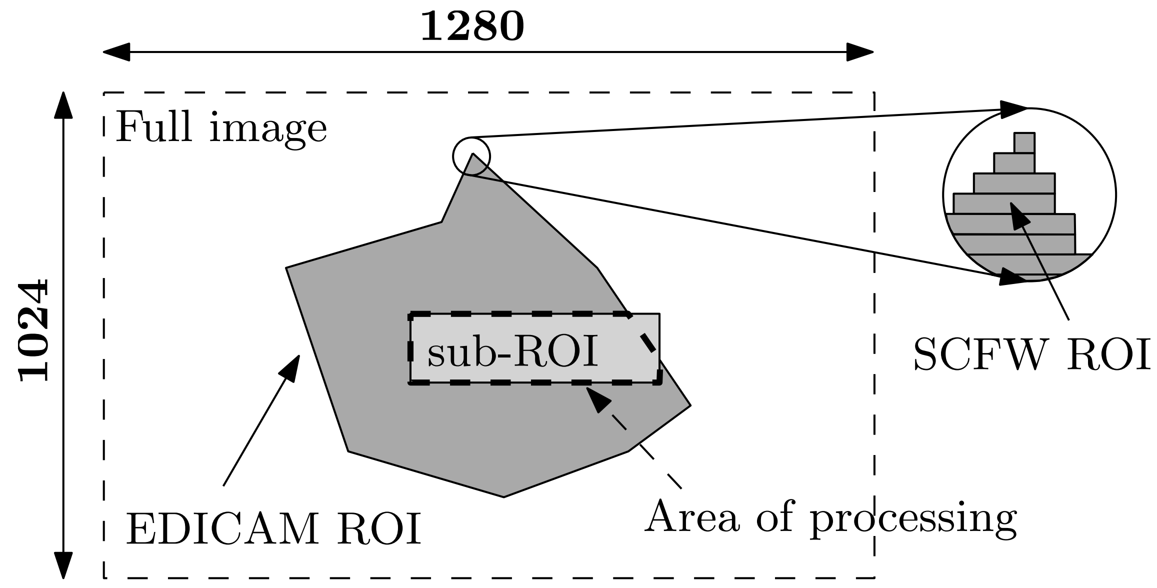

Figure 2.2. *Illustration of ROI related units.*

This entire description is sent to the *SM* when a *readout* is generated by a *ROIP*. After the transmission of the *readout request* has been finished, *SM* performs the interpretation of the complex *EDICAM ROI* descriptor, and performs the *readout* of each rectangular area (*SCFW ROI*) sequentally. The image data sent by *SM* can be processed by *IPCU* and transmitted to the *PC*.

A *ROIP* starts to initiate *readouts* after it has been activated by the event handling mechanism of *EDICAM* or by the *PC*. The possible operation modes are illustrated in figure 2.3., and explained in the following.

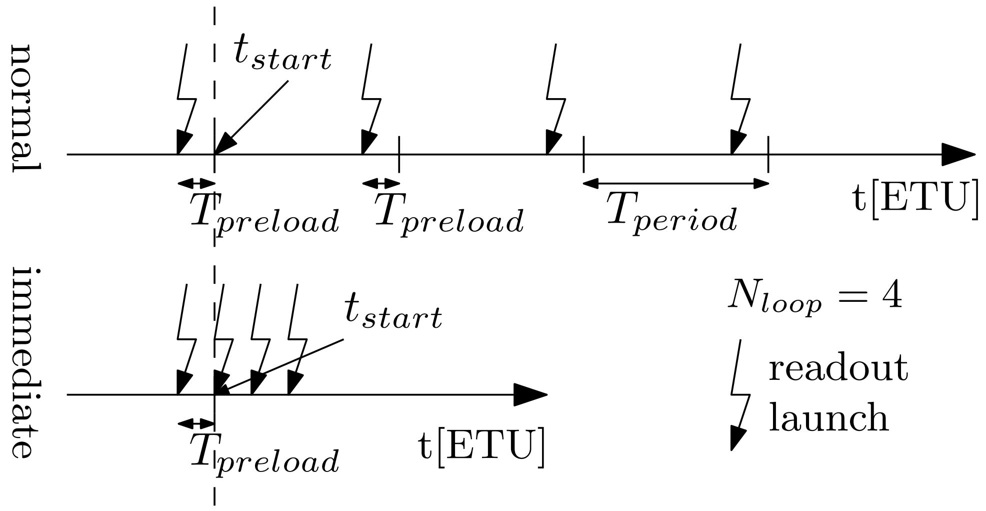

Figure 2.3. *Difference between normal and immediate ROIP timings.*

Two kinds of *ROIPs* are defined: *immediate* and *normal*. A *normal ROIP* generates the first *readout request* before the start time by $T_{preload}$. The start time (t_{start}, sampling time of the first *readout*) is available in the description. Each additional *readout* launch is delayed by T_{period}. The number of executed *readouts* is defined by N_{loop}. If *immediate* mode is specified, *readout requests* will be generated as fast as possible after the launch time of the first *request* arrives ($t_{start} - T_{preload}$). *Normal readouts* wait for the sampling time in *SM* ($T_{preload}$ affects only the launch time), while the *immediate* ones are performed as soon as possible.

There are differentiated *persistent* and *non persistent ROIPs* as well, regardless of the *immediate* and *normal* attributes. *Normal non persistent ROIPs* discard *requests* examined beyond the corresponding sampling time, which can be caused by e.g. a too late activation. If the sampling time of the current *request* (after some discards) is greater than the current *ETU time*, it will be launched. In contrast, *immediate non persistent ROIPs* cancel all *requests* if the first cannot be generated in time. In case of *persistent* operation, every *requests* will be launched in both modes.

Since *persistent* property is interpreted differently by *immediate* and *normal ROIPs*, the former is not equal to the latter with T_{period} set to 0. Furthermore, the sampling time in *SM* of *readouts* generated by only *normal ROIPs* are checked again (they can be discarded also in the *SM*), while the *immediate requests* which reach the *SM* are performed in any event (practically *immediate requests* are performed independently from sampling time).

There is one more important parameter which describes the operation of *ROIPs*, namely the trigger mode. *ROIPs* are slightly affected by this attribute, only the $T_{preload}$ parameter is influenced. The real difference is caused in the operation of the *readout* module in *SM*, which is introduced in 2.4.2 (*Readout operation*).

2.3.2 Event description

The high level control of *EDICAM* is performed by a computer (*HOST*). The *HOST* continuously monitors the state of the system, and changes the operation if necessary. Since *EDICAM* requires very low latency between a status change and the corresponding control reaction, an internal control mechanism has been applied. This concept is implemented by the so called *events* which are introduced hereafter.

Events realise low latency (immediate), programmable control of the *EDICAM* operation. They use multiple binary inputs which are combined with predefined and programmable binary operations. The single bit result of the operations is considered as the state of the *event*. When an *event* becomes active or inactive, multiple actions can be performed. These actions are the above mentioned low latency control reactions.

Events are defined by the so called *event* inputs, operation and actions:

Event inputs

Each *event* has a certain number of *event inputs*. Each *event input* of each *event* are driven by the following sources:

- Certain properties of the image data.
- State of external inputs and other events.
- *ETU* time.
- Inputs controlled by the *HOST* via *PCIe*.

Event operation

The following operations can be performed on the *event* inputs:

- Negation of arbitrary inputs.
- Combination with binary *AND* operation.
- The result of the *AND* operation can be negated.
- The state change of the output (the negated *AND* operation) can be delayed by a certain amount of time in *ETU*.

Event actions

There are predefined actions in *EDICAM* which can be associated with certain events. An event can control even more actions. The predefined actions are the following:

- Switch a *ROIP* state to active or inactive.
- Control a *ROIP* active state.
- Send *ROIP* trigger to *SM* to start a triggered *readout*.

- Send *exposure* trigger to *SM* to start a triggered *exposure*.
- Assert or deassert a pin when the *event* becomes active.
- Control or invert the state of a pin.
- Send interrupt to *HOST*.

These actions can be triggered by various *event* state conditions: rising edge, falling edge, high level and low level.

Image data processing

Certain *event* inputs are the result of image data processing. The image data is sent by *SM* as a result of a *readout*, and the pixels are processed by the *IPCU*: minimum, maximum and total intensity is calculated. These 3 parameters are compared against a threshold level, and the single bit results of the comparisons are part of the inputs of *evenets*.

ROIs enable *EDICAM* to perform the *readout* only of subparts of the full image (see illustration in figure 2.2.). The extension of the processing of the full image can be further limited (beyond *ROIs*) by the so called *sub-ROIs* (sROI). *sROIs* are rectangular areas that fully or partially cover a *ROI*. An *sROI* defines a rectangle relative to the upper left corner of the full image. An *sROI* might extend over the *ROI* area e.g. if the *ROI* has arbitrary shape. In this case only data of the common territory of the *ROI* and *sROI* are processed.

2.4 SM related parts

The specification contains subparts related to *readouts* and *exposure* which can be associated with the *SM*. These parts are described in this section.

2.4.1 Exposure control

The sensor module can perform image exposure periodically. *Readout* is not coupled with exposure, so it can be done on any part of the diode array at any time independent of the exposure. The exposure control can be either in *IDLE* state (no exposures are running), *ARM* state (parameters are set, waiting for start time) or *RUN* state when periodic exposures are running as shown in figure 2.4. *RUN* state can last for either an infinite time or can return to *IDLE* after a predefined number of exposures have been performed. An incompleted exposure sequence can be stopped by the *IPCU*.

After a short initialization period, the exposure control at *SM* gets into *IDLE* state. After the exposure parameters have been set, an exposure start command is sent from the *IPCU* to *SM*. The exposure state of *SM* changes to *ARM*. The actual exposure sequence starts only when the time of the first exposure is reached or the triggered exposure receives a trigger.

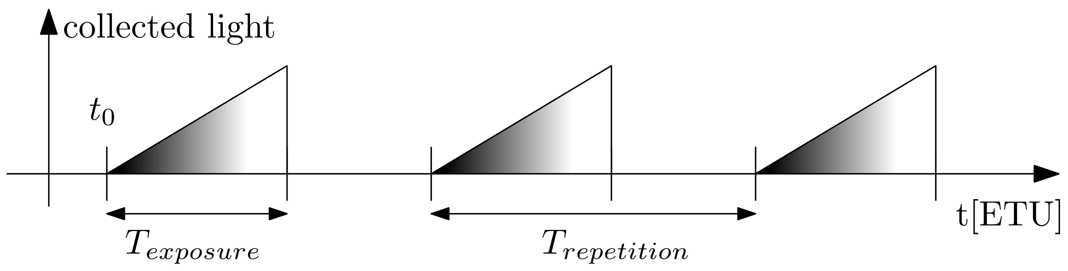

Figure 2.4. *Exposure sequence in SM.*

Exposure can be described by the following parameters which can be set by the user in the *SM* register table (see figure 2.4.):

- Start time instant of first exposure in *ETU* (t_0) or start on external trigger. The external trigger is received by the *IPCU* and forwarded to *SM* via the 10G link.
- Exposure time interval($T_{exposure}$).
- Repetition period ($T_{repetition}$).
- Number of exposures or 0 if infinite loop is desired (N_{loop}).

2.4.2 Readout operation

This subsection describes that parts of the *ROIP* concept in *EDICAM* which are not related to the *IPCU* exclusively. The following details are based on subsection 2.3.1, but a short summary of the relevant parts is presented as well.

SM performs elementary *readouts*: The *ROIP* cores in *IPCU* translate the periodic image read sequences defined by the *PC* into individual *readouts*, so all *readouts* are triggered one-by-one by the *IPCU*. The packet of informations which describe a *readout* is referred to as a *readout request*. The basic *readout* operation in the sensor module consists of the following steps as shown in figure 2.5.:

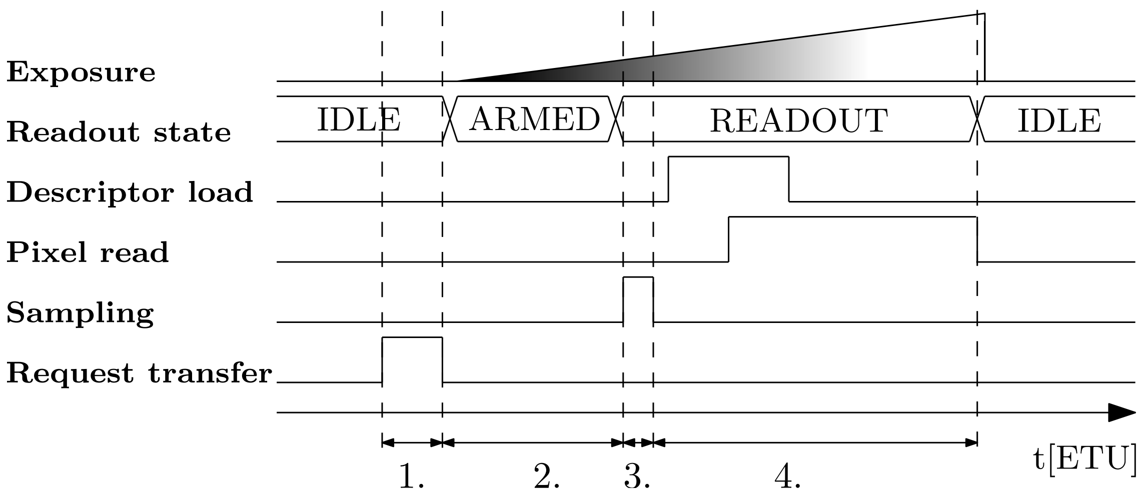

Figure 2.5. *Timing diagram of a basic readout sequence in SM.*

1. Transfer of a *readout request*: The *readout request* is sent from the *IPCU* to the *Sensor module*. This contains information on what should be read out and when the *readout* should start.

2. Waiting for sampling time: *SM* is waiting until the sampling time instant specified in the *request*. The sampling time is defined in the *request* as a time instant in *ETU*, or can be determined by an external trigger.

3. Sampling: Sampling impulse is generated towards *SCFW*.

4. *Readout*: The descriptor of the first *ROI* related to the *actual request* is loaded to the sensor at first. The digitized pixel information has to be read continuously from the sensor meanwhile the parameters of rectangular *ROIs* are loaded to the sensor. The image data is sent onward to the *IPCU*. The procedure is finished when the last rectangular *ROI* of the area defined by the *ROIP* has been also read.

With regard to the readout sequence, the following restrictions have to be considered:

- No sampling can take place during reading samples from *LUPA*1300 (pixel read).

- The analog sampled image is available for pixel read for approximately 2 ms. Multiple *ROI* packets can make use of the same sampled analog data, but the data is lost after 2 ms. Beyond this interval, the sampled analog values have to be considered to be invalid.

The previous enumeration describes the *readout* operation from a high level point of view. The *EDICAM* specification includes also some hints related to the implementation of the *readout* operation which are presented in the following.

The *SM* has three readout states, namely *IDLE*, *ARMED* and *READOUT*. *SM* is in *IDLE* state by default, while it waits for the download of a *readout request*. When a *request* arrives, *SM* enters *ARMED* state. In this state, *SM* waits for the sampling time specified in the *request*. As soon as the *ETU time* in *SM* reaches the sampling time instant, a state transition occurs to *READOUT* and *SM* starts digitizing data and downloading rectangular *ROI* descriptors. Meanwhile, the image data are sent continuously to the *IPCU*. After pixel read and transmission of data have been finished, *SM* checks whether a new *request* is already available in the *request* buffer. If it is, *SM* will enter *ARMED* state otherwise it returns to *IDLE* state.

The *ETU* time when the last sensor sample is taken is registered in a *SM* internal register. This register is cleared after a certain time has passed which can be set as well. If the required sampling time in a *readout* command equals the time of the last sample, then without a new sensor sampling a *readout* will be performed. There are also conditions when sampling cannot be preformed by the *SM*. In such cases the *ARMED* state is aborted and *SM* returns to *IDLE* state. Sampling is interrupted when the required sampling time of a *ROI* has already been passed or when a *clear readout* command is received. Regardless of the conditions, there is always a *readout* command sent form *SM* to *IPCU* in response to a *ROI* download. If no data are available, the header of the command will contain an error code.

SM has storage capacity for two *readout requests*. These so called *readout request registers* are filled by the *IPCU*. The active *request* is the one which determines the current *readout*. As soon as the *readout* has been finished, the role of the two registers are swapped and the previously active *request register* becomes free, waiting for the next *request*.

Handling of the sampling time instant depends on the following three flags of the *ROIP* description which are sent to *SM*:

- *immediate*: The sampling is done immediately as soon the *SM* has been entered into *ARMED* state.

- *persistent*: If this flag is set, the readout process will read the image even if the indicated sampling time has passed. Without setting this flag, the readout command will be dropped if the sample time is passed. This behaviour is applied to both triggered and timed *ROIs* as well.

- *triggered*: If this flag is set, the *readout* will wait for a readout trigger command from the *IPCU*, and the sampling will not be taken at the indicated sampling time instant. However, if the *persistence* flag is not set, the *readout* is cancelled when the indicated sampling time has passed. Triggered *ROIPs* suspend the sensor operation for an unknown time therefore practically no other *requests* can be processed.

2.5 SM-IPCU interaction

This section introduces a typical *readout* scenario. The scenario consists of an *exposure* setup followed by two *readout* requests. The second *request* is cancelled before the corresponding sampling time instant.

The description is focused on the interaction between *IPCU* and *SM*. The state transitions of the *exposure* and *readout* operations implemented in *SM* and the corresponding actions are illustrated in figure 2.6. and detailed in the following. Note that these so called operations are the high level descriptor of the *EDICAM* operation.

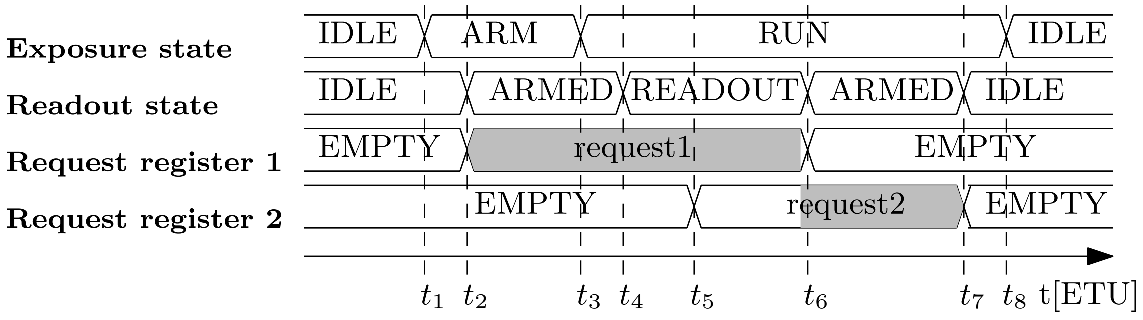

Figure 2.6. *State transitions of SM submodules during a basic readout.*

SM is in indifferent state before t_1: Both *exposure* and *readout* are in *IDLE* state, and the *request registers* are empty. The operations are waiting for parameters and a start trigger. A command which contains *exposure* related paramteres arrive at t_1, and hence *exposure* operation steps into state *ARM*. During this state, *exposure* is waiting for the time instant of the first *exposure* which arrives at t_3. *Exposure* is in state *RUN* until t_8: It transitions back to *IDLE* after the last *exposure* defined by the parameters has been performed. *Readouts* are performed during *exposures* are running (between t_3 and t_8).

The *readout* operation steps into *ARMED* state when a *readout request* is launched by a *ROIP* at t_2 (*request*1). The *request* is stored in the first *request register*. Since both registers were empty before the arrival of the *request*, the new *request* will be *active* immediately. The *sampling time* of the first *request* arrives at t_4, and hence *readout* operation steps into state *READOUT*: sampling, *ROI* download and image data transmission towards the *IPCU* is carried out.

Readout operation steps back to *ARMED* state, and *request*2 (launched at t_5) in the second *request register* becomes active when the data transmission related to *request*1 is finished (t_6). *Readout* operation is waiting for the sampling time of *request*2 between t_6 and t_7. *ARMED* state is followed by *IDLE* state instead of *READOUT* state (t_7), because a clear *readout* command occurs at t_7. The *request registers* become empty after the corresponding *readouts* have been performed (t_6, t_7), so further *requests* can be launched by the *IPCU*.

Chapter 3

System level design

This chapter presents system level design considerations in the *EDICAM* firmware. The high level block diagram of *EDICAM* is shown again in figure 3.1., because it exhibits the structure at system level. System level design considerations deal with clock domain- and reset structures as well as interfaces between top level modules. The presented results are trade-offs based on different types of aspects, including operating frequency, development time and resource utilization. The block diagrams and high level descriptions of the *user logic* modules are also introduced. The operation of the corresponding submodules is summarized in Chapter 4.

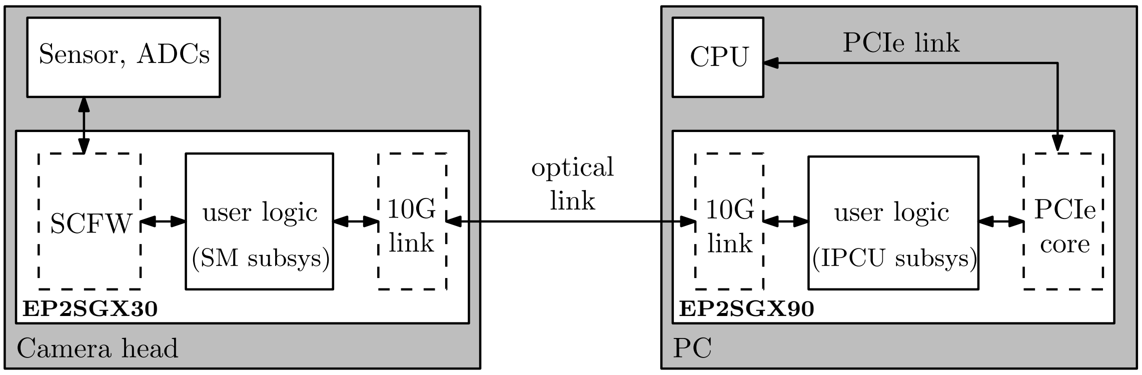

Figure 3.1. *High level block diagram of EDICAM.*

3.1 Sensor Module

The Sensor Module (SM) is located in the *Camera head*. This part of the firmware is tagged *EP2SGX30* in figure 3.1. which refers to the type of the corresponding *FPGA*. This section addresses connections between the high level modules, such as *SCFW*, *user logic* and 10*G link* among others.

3.1.1 Clock and reset structure

The clock and reset structure of the *SM* can be seen in figure 3.2. The incoming clock signals of this module are $clk40\_i$ and $xaui\_refclk\_i$. The latter is used only by 10*G* *link* to drive the *XAUI* interface of the optical link, while the 40 MHz incoming clock and its 100 MHz transformed version (referred in the following as *clk*40 and *clk*100) is used by other *SM* modules. *clk*100 is derived by a *PLL*. Since this *PLL* is reset by the *Power supervisor* and the corresponding *locked* output feeds the reset system of *SM*, the correct startup is guaranteed.

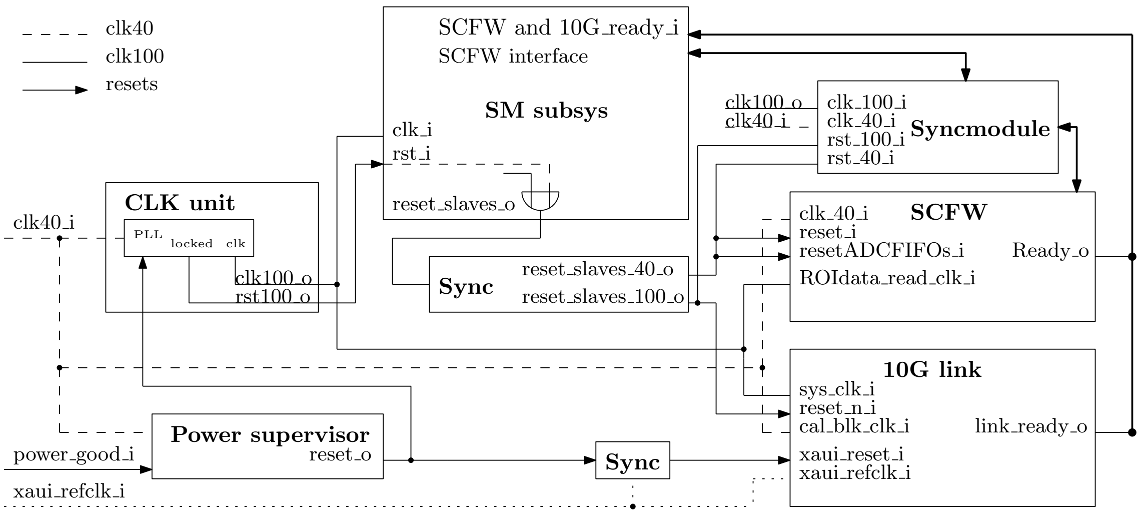

Figure 3.2. *Clock and reset structure of SM*

The applied clock frequencies are determined by the *EDICAM* specification and constraints related to reused modules. *EDICAM* can perform the *readout* of complete images up to 400 times per a second with even $1280 * 1024$ pixel resolution. Since each pixel produces 12 bit information, to provide this performance, a transmission speed greater than 5.86 Gbps has to be supported. Because the *data interface* of *Link transmitter* has a width of 64-bit, the necessary operating frequency of this interface is approximately 92 MHz. The selected 100 MHz for the data path can be produced easily by a *PLL* from the 40 MHz clock, and it ensures some spare bandwidth as well. These clocks are considered *unrelated* in *SM*. To sum up, the *SM subsys* (this module implements the functionality related to *EDICAM*) is fully, while *SCFW* and 10*G link* are partly controlled by *clk*100. A guideline for clocks is to operate the modules on the smallest feasible frequency, which results in *looser timing requirements* and *reduced area consumption*.

ROI parameter and *control* interfaces of *SCFW* are controlled by the 40 MHz clock input of *SCFW*. This clock has a fixed period, so synchronisation is necessary between *SM subsys* and *SCFW*. This is performed by *Syncmodule*. The *Image data* port of *SCFW* is fed by another clock input which allows the use of clocks with arbitrary period. To avoid synchronisation in the image datastream, this port is connected to the same *clk*100 as *SM subsys*. The relevant interfaces of 10*G link* from the *SM subsys* point of view (*Receive* and *Transmit* interface) are provided with unconstrained clock input as well, so these ports are also clocked by *clk*100.

The operation of the reset structure of the *SM* is demonstrated via figure 3.3. The following description focuses on the reset signals which are controlled by *SM subsys*. Before t_1, the states of *SM* after a power-up can be seen. The whole module is reset until

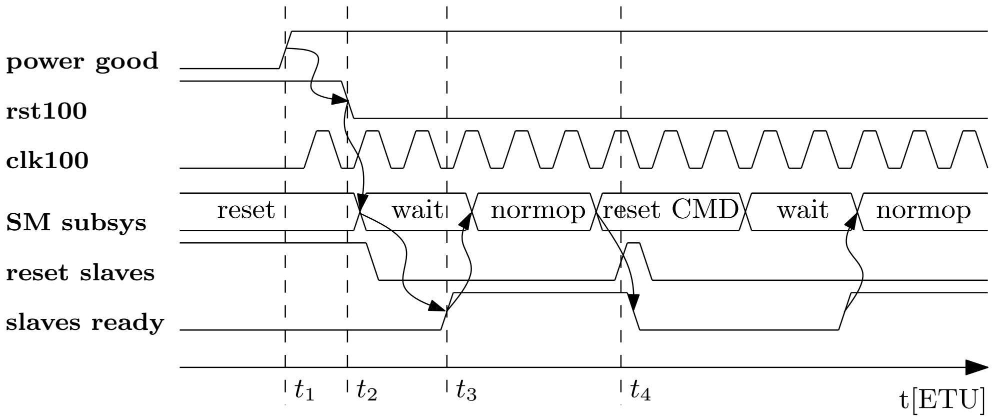

Figure 3.3. *Power-up and reset command timing sequence.*

the power supply is not stabilized. This is indicated by *Power supervisor* at t_1 by the assertion of the signal *power good*. Between t_1 and t_2, the *PLL* tries to lock on *clk*40. The *PLL* is locked when *clk*100 is stabilized and hence signal *rst*100 transitions to 0. After t_2, *SM subsys* leaves state *reset*, and so deasserts the reset signal of the slaves (10*G* and *SCFW*). The slave modules have to be ready before *SM subsys* transitions to its normal operation (*normop*, t_3). There is a reset command which restarts the operation both of *SM subsys* and the slaves. Such an event can be seen at t_4: After a short reset pulse is generated on *reset slaves*, the same sequence starts again as at t_2.

Finer division of the clocks of the *SM* would be possible with even slower clocks, however it would result in the unnecessary increment of the development time. The presented results are representing a properly constrained system which complies the specification without any exertion related to timing closure or resource utilization.

3.1.2 Interface synchronisation

As described in the previous subsection, the *control* and *ROI parameter* interfaces of *SCFW* are in the 40 MHz domain. A synchronisation module has to be connected to these ports, because *SM subsys* is in the 100 MHz domain. *Syncmodul* emulates the behaviour of the interfaces related to *SCFW* towards *SM subsys*, however, in the faster domain. The structure of *Syncmodul* can be seen in figure 3.4.

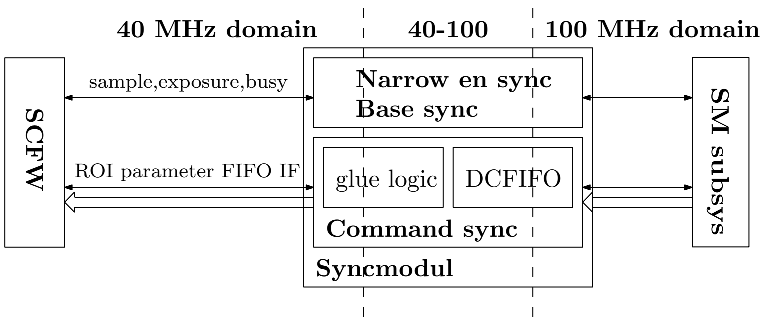

Figure 3.4. *Block diagram of Syncmodul.*

The synchroniser can be divided into two different parts. *ROI parameter interface* is a FIFO interface with parallel control, status and data signals, while the signals *sample*, *exposure* and *busy* are independent single bit control lines. This *FIFO* interface can be transmitted via a *DCFIFO* expanded with some glue logic which ensures the proper behaviour of the control signals of the *parameter* port. This functionality is implemented by *Command sync*. The single bit lines are synchronised simply by base and enable[1] synchronisers. A base synchroniser consists of two serially connected *FFs*.

3.1.3 High level user logic introduction

This subsection presents a high level overview of *user logic* located in *SM* (see figure 3.1.). The overview consists of a block diagram which represents the main components of *user logic* along with the corresponding interfaces (see figure 3.5.). The introduction of interfaces facilitates the handling of the complexity of the top module. Section 4.1 presents the details of each module and interface which have not been described in the thesis yet.

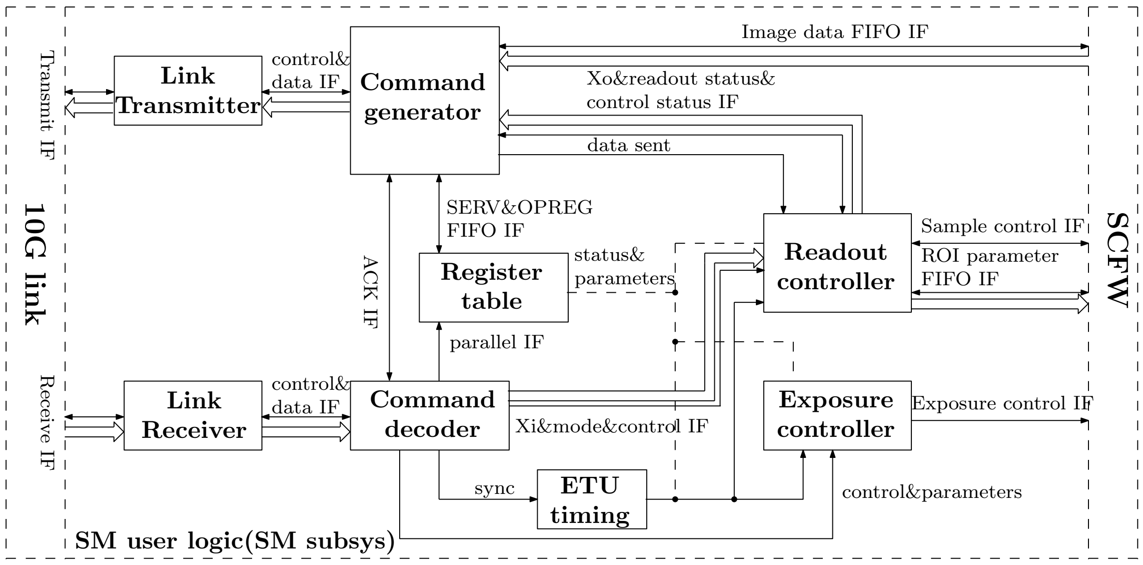

Figure 3.5. *Block diagram of user logic in SM.*

SM subsys is connected to the 10*G link* via *Link Receiver* and *Link Transmitter*. *Link Transmitter* receives data packets sent by internal modules via a *QW* stream. The packets are completed with a cyclic redundancy check (CRC) code, and the boundary *QWs* are marked by the appropriate framing singals of *Transmit IF*. *Link Receiver* processes the incoming commands: The lenght and the *CRC* code of the received packets are checked. The appropriate error signals are asserted upon failure.

The interpretation of the commands received by *Link Receiver* is performed by *Command decoder*. This module controls the internal modules of *user logic* as required by the current command. Some parameters are transmitted via the *parallel IF* (interface) toward *Register table*, so they reach their destination indirectly. *Register table* also receives status information from the other modules, and provides it for *Command generator* via *FIFO* interfaces. The organization of the communication in the opposite direction is implemented

[1]The applied enable synchroniser (*Narrow en sync*) is introduced in subsection 4.3.3.

by *Command generator*. This module provides various ports which allows the other units to simply launch commands and optionally transmit data toward *IPCU*.

The direct control of *SCFW* is perfomed by *Exposure-* and *Readout controller*, but note that the image data output is connected to *Command generator*. *Readout controller* provides not only status information related to the *readout* toward *Command generator* but also the *ROI description* (it is sent back to *IPCU*). These descriptors are stored in the *read request buffer* of *Readout controller* (see subsection 2.4.2) which is fed by *Command decoder*.

Since the both sensor related modules perform timing based on the system time, they are connected to *ETU timing*. The timing unit can be loaded during operation because of the different frequency uncertainty of the oscillators in *Camera head* and *Altera* development board.

3.2 IPCU

Image Processing and Control Unit (IPCU) is located in the *Altera* development board. This part of the firmware is tagged as *EP2SGX*90 in figure 3.1. which refers to the type of the corresponding *FPGA*. This section addresses connections between the high level modules, such as 10*G link*, *user logic* and *PCIe core* among others.

3.2.1 Clock and reset structure

The clock and reset structure of the *IPCU* can be seen in figure 3.6. The *Application interface* (*APP IF*) provided by the modified example application related to the *PCIe core* is illustrated separately.

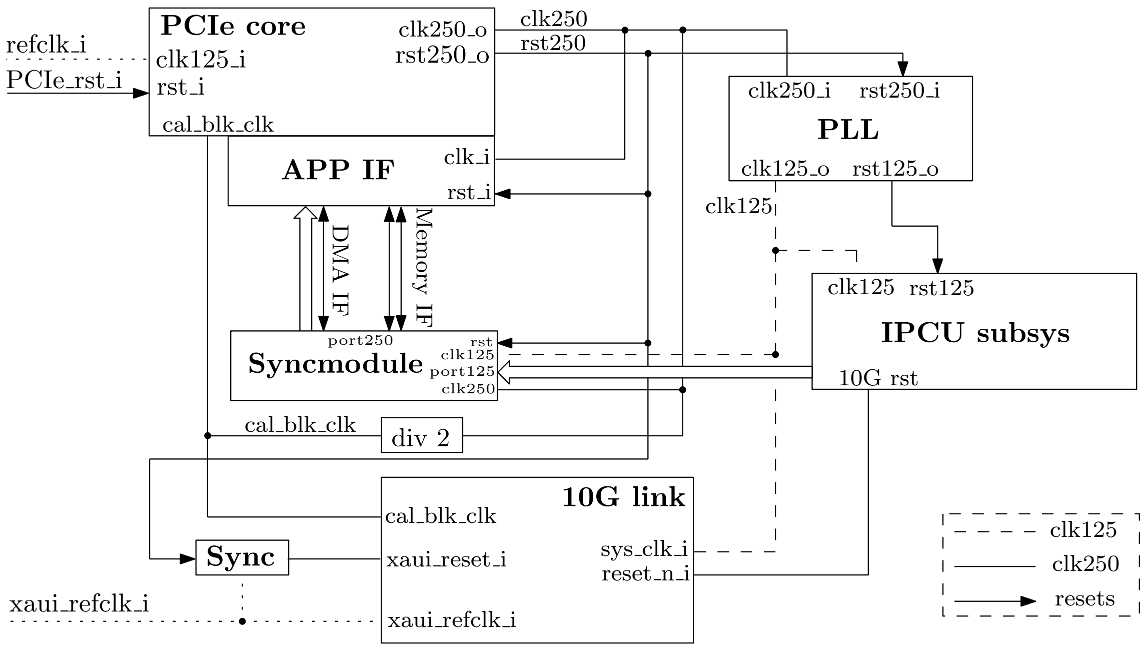

Figure 3.6. *Clock and reset structure of IPCU.*

The incoming clock signals of this part of the firmware are $refclk\_i$ and $xaui\_refclk\_i$. $xaui\_refclk\_i$ is used only by $10G$ *link* to drive the $XAUI$ interface of the optical link. $refclk\_i$ with a frequency of 125 MHz is used by the $PCIe$ *core*: The frequency is duplicated, and the generated clock signal with a frequency of 250 MHz ($clk250$) is used by the logic inside the $PCIe$ core. cal_blk_clk is the common calibration clock of the gigabit transceivers, and is generated from $clk250$ by a simple $D\ FF$.

Since SM transmits image data at a maximum speed of 100 MHz, and the data processing is performed by fully pipelined parallel structures, an operation frequency of 100 MHz is satisfactory in $IPCU$ *subsys*: A PLL generates a clock with 125 MHz ($clk125$) from the output clock of the $PCIe$ *core*. The 125 MHz frequency overfulfils the minimal criterion, but provides benefits in data transmission between clocks $clk125$ and $clk250$ (these clocks are related). The relevant interfaces of $10G$ *link* from the $IPCU$ *subsys* point of view (*Receive* and $Transmit\ interface$) are provided with unconstrained clock input, so these ports are clocked by $clk125$ as well.

The *Read-*, $Write$ *memory* and $DMA\ FIFO\ WR\ interface$ of $PCIe$ *core* (see subsection 1.4.3) are controlled by $clk250$, and hence synchronisation has to be performed between the $PCIe$ *core* and $IPCU$ *subsys*. *Syncmodule* is connected to these high speed interfaces, and performs the data transmission between the two domains via $port250$ and $port125$. The module consists of $DCFIFOs$ expanded with some glue logic.

The operation of the reset structure is shortly summarized. First, the $PCIe$ *core* has to be initalized after *power on reset*, which deasserts the reset signals of the *Syncmodule*, PLL and $10G$ *link*. $IPCU$ *subsys* starts to operate after the PLL has been locked, and deasserts the user reset signal ($reset\_n\_i$) of $10G$ *link*.

3.2.2 High level user logic introduction

This subsection presents a high level overview of *user logic* located in $IPCU$ (see figure 3.1.). The overview consists of a block diagram which represents the main components of *user logic* along with some of the corresponding interfaces (see figure 3.7.). The details of the submodules are presented in section 4.2.

I have contributed to the development of $EDICAM$ *subsys* only partially, and hence section 4.2 presents only the relevant units (RDP and *Event processor*) in detail. A short description of the other submodules is presented in this subsection as well.

The core unit of $IPCU$ *subsys* is the *Readout command generator* (RCG) which implements the $ROIPs$ introduced in subsection 2.3.1: RCG is responsible for generating periodic readout *requests* based on the parameters provided by the *Register table*.

Commands are initiated also by CMD *requesters*. This unit consists of submodules which are generating periodic requests toward *Sensor module command queue* (SMCQ): clock sync- and status requests. The request period can be adjusted in the *Register table*.

The various command requests are received by $SMCQ$. After the arbitration has been performed between the requests, the commands are generated with proper framing: The necessary pieces of information are collected from the appropriate submodules, e.g. *Register*

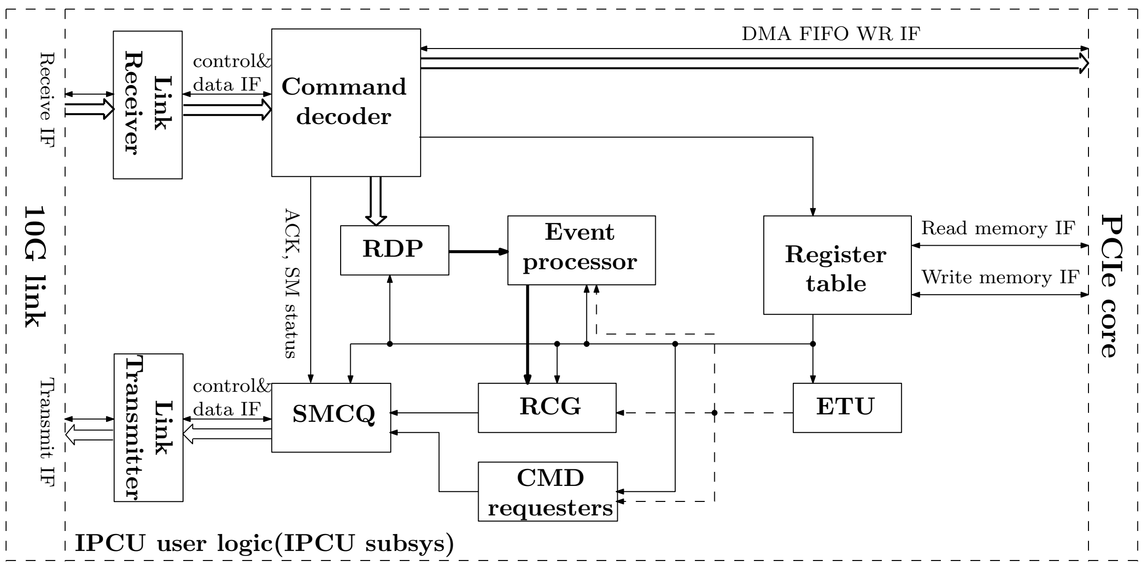

Figure 3.7. *Block diagram of user logic in IPCU.*

table. The commands are sent via *Link Transmitter*.

Data traffic on the 10*G link* in the opposite direction is decoded by *Command decoder*. Every command is acknowledged by *SM*. When such a command arrives, *Command decoder* indicates this event toward *SMCQ*. *SMCQ* sends the next command after the previous has been acknowledged. The status of *SM* can be quoted, and the status bits are transmitted to *Register table*. If a command which contains image data arrives, data bits are sent to both *RDP* and *PC* (latter via *DMA FIFO WR IF*).

The *processing* and *control* features of *IPCU* are implemented by *ROI data processor* (RDP) and *Event processor*. *RDP* processes image data (minimum, maximum, intensity), and feeds certain inputs of the *Event processor*. *Event processor* asserts trigger signals related to several actions. The operation of *Event processor* is defined by its inputs and configuration. The configuration is located in the *Register table*.

Since some *ICPU* related modules carry out timing (*Event processor*, *RCG* and *CMD requesters*), *ETU timing* is instantiated as well.

The operation of the above introduced submodules can be controlled and observed by the *HOST* via *Register table*.

Chapter 4

Implementation of submodules

This chapter presents the details of the implementation and operation of the submodules introduced in subsections 3.1.3 and 3.2.2. The submodules which are not the outcome of my development work are omitted. The description focuses on the high level interfaces and main features of the submodules. The objective of this part of the thesis is to provide a short overview of the structure of this complex system.

4.1 SM related submodules

During my work, the *readout* related modules (*Exposure controller* and *Readout controller*) along with the *Command generator* were developed that are related to the *Sensor Module*. The details of these units are presented hereafter. The simultaneous observation of figure 3.5. (*SM subsys* blockdiagram) is highly recommended in order to simplify the understanding of this section. The omitted modules are the *SM Register table* and *Command decoder* which are not part of my work.

4.1.1 Exposure controller

The *exposure* signal of *SCFW* is directly controlled by *Exposure controller*. This unit provides a simple interface (see figure 4.1.) which can be used by the control module of *SM* to set the parameters of the epxosure (see subsection 2.4.1).

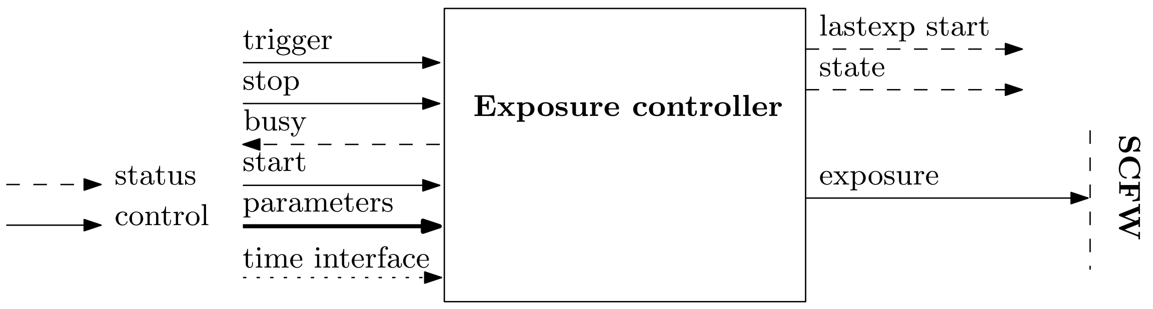

Figure 4.1. *Interface of Exposure controller.*

An exposure sequence can be launched when the *busy* signal of *Exposure controller* is not asserted, which indicates that an exposure sequence is running. Unless *busy* is kept high, the operation starts immediately after the assertion of *start* which registers the input parameters (the assertion of *start* is missed if *busy* is kept high). These parameters are the same as the ones introduced in the high level *EDICAM* specification (exposure-, repetition time, number of exposures and the selected operating mode with start time).

In the first step of the operation (*ARM* state) *Exposure controller* is waiting for the start event. In normal mode, the start time has to arrive, in triggered mode, the input *trigger* has to be asserted beyond *start*. *Exposure controller* steps into state *RUN* if immediate mode flag is set among the parameters. At the beginning of every exposure, the current *ETU time* is stored which, along with the current internal state (*IDLE*, *ARM*, *RUN*), is available at the output of the unit. This information is contained in the corresponding data stream.

The operation can be aborted by the assertion of *stop* any time. The input is registered and *Exposure controller* transitions to *IDLE*. In state *RUN*, not the actual but the next cycle will be aborted. Another exposure can be launched only after the current one is completed or aborted.

The module consists of a state machine, a counter which tracks the number of the remaining expositions and a counter which monitors the length of the exposures and reset cycles.

4.1.2 Readout controller

Readout requests generated by *IPCU* are executed by the *Readout controller*. The implementation is based on the parts of the specification presented in subsections 2.3.1 and 2.4.2.

The introduction of this module is started with the description of the provided interfaces which are the following: *Xi interface*, *mode interface*, *control interface*, *time interface*, *control status interface*, *Xo interface*, *readout status interface*, *ROI parameter FIFO interface* and *sample control interface* (see figure 4.2.).

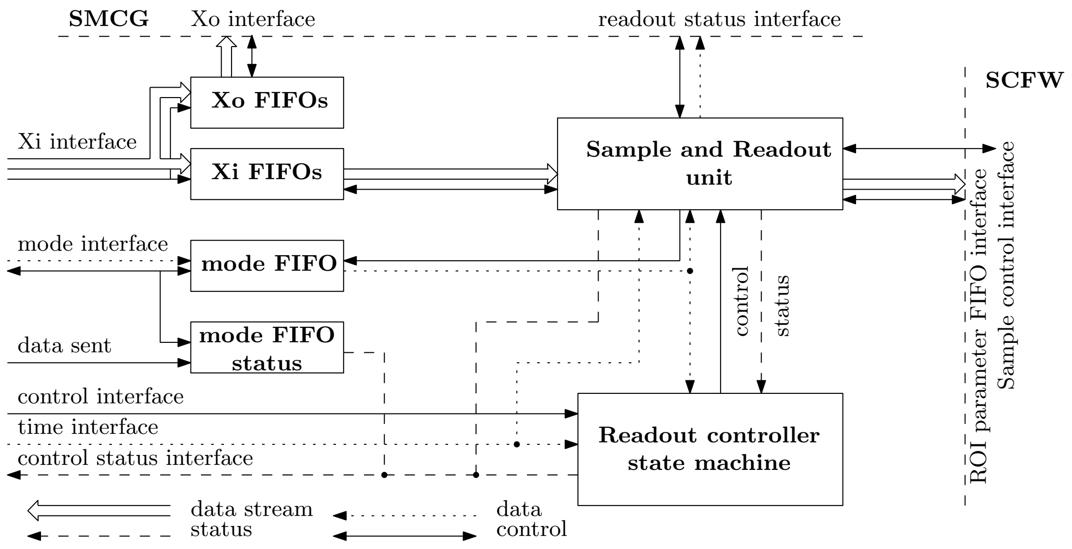

Figure 4.2. *Block diagram of Readout controller.*

Parameters in *readout requests* can be transmitted on $Xi\ interface$ and $mode\ interface$ which are implemented as simple $FIFO$ interfaces. These interfaces are referred to as request buffers in the specification. $Xi\ interface$ is connected to $FIFOs$ which have enough capacity to accommodate the row descriptors of two *requests* with up to 1024 row and arbitrary shape. The $FIFOs$ fed by $Xi\ interface$ are duplicated, because $SMCG$ sends also the shape descriptors along with the image data to the $IPCU$ as well. The other *request* parameter interface is the $mode\ interface$ which is related to a $FIFO$ with 2 depth. This port has to be filled with parameters like sample mode (*immediate*, $persistent$, *triggered*), sample time, ROI type (arbitrary/rectangular) and number of rows. These parameters are necessary to determine the time of sampling and to reproduce the ROI parameters required by $SCFW$. The *readout* is influenced by the $control\ interface$ which consists of clear and trigger signals. Readouts waiting for the sample time or for the arrival of a trigger can be canceled by the clear signal before the *readout* has not been started yet. Some state information are accessible on $control\ status\ interface$, namely the time of the last sample, the ID of the ROI related to the currently processed *request*, the status of the *request* buffer and the state of *Readout controller* ($IDLE$, $ARMED$, $READOUT$) itself.

After the ROI parameters in the *readout request* have been transmitted to $SCFW$ on $ROI\ parameter\ FIFO\ interface$, or the *readout* has been canceled, the status of the *readout* can be received on $readout\ status\ interface$: time of sampling, time of exposure, flags indicating whether readout has been canceled (timeout or stopped) and the other pieces of information which were also available via $mode\ interface$.

This subsection introduces the internal structure and operation of *Readout controller* as well, which is described in the following. The main components of *Readout controller* can be seen in figure 4.2. These components are the $FIFOs$ connected to the interfaces described in the previous paragraphs, a state machine (*Readout controller SM*) and a separated submodule (*Sample and Readout unit*). The operation of *Readout controller SM* is summarized in figure 4.3. and detailed hereafter.

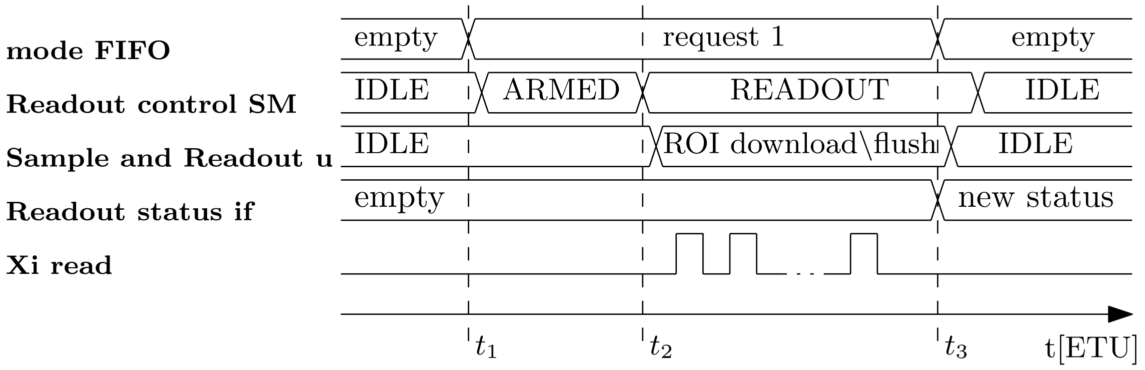

Figure 4.3. *State diagram of submodules in Readout controller.*

If a *readout request* is loaded into *mode FIFO* (t_1), *Readout controller SM* will determine the launch or abolition of the *readout* ($ARMED$ state). The operation is determined by the following parameters and signals: clear and trigger signals, sample time, $persistence$ and *immediate* mode. After the start time or a trigger arrives, or the *readout* is canceled (t_2), *Sample and Readout unit* is started. This unit performs the transmission of *ROI* descriptors to *SCFW* (note that this is only a part of the *readout* procedure) or simply flushes the contents of *Xi FIFOs* related to the active *request* in *mode FIFO*. The latter is performed when the *readout* is canceled. The results of the operation is stored in *readout status FIFO* when *Sample and Readout unit* is finished (t_3). *Readout control SM* is in *READOUT* state during *Sample and Readout unit* is in operation. Before *Sample and Readout unit* transitions back to *IDLE* state, it clears the parameters of the currently processed *request* stored in *mode FIFO* (t_3). This *request* is referred to as *active request* in *EDICAM*: e.g. in figure 4.3. *request* 1 is the active *request* between t_1 and t_3.

The *IPCU* is allowed to launch a *request* if the request buffer is not full (*mode FIFO* and *Xi FIFO*). The fullness of the buffer is monitored by *mode FIFO status*. This unit is based on a counter which is incremented when a write occurs on *mode interface* (transmission of a new *readout request*), and is decremented when *data sent* input is asserted. *Data sent* signal is kept high for one period by *SMCG* when the package containing the image data of the active *readout request* is loaded completely to *Link Transmitter*. Since this data package contains also the *ROI* descriptor which is accessed via *Xo FIFOs*, the next *request* can not replace the active *request* in *Readout controller* before all of the row parameters (related to the active *request*) from *Xo FIFOs* are read by *SMCG*.

Sample and Readout unit

Sample and Readout unit is a submodule located in *Readout controller*. The operation of this unit is based on *Xi FIFOs*, *mode FIFO* and *time interface*, and controlled by *Readout controller SM* (see figure 4.2.). The module can perform two different kinds of operations. If the abolition of the active *request* is determined by *Readout controller*, this submodule simply flushes the appropriate number of elements from *Xi FIFOs*, clears the active request located in *mode FIFO*, and writes the readout status to *readout status FIFO* with the stopped/timeout flags set. This way, no communication is performed with *SCFW*.

In contrast, if a sample is triggered by *Redout controller SM*, the *ROI* descriptor will be loaded into *SCFW*. Unless otherwise noted, a sampling will precede the transmission of the first *ROI* descriptor (if the sample time matches the previous, no sampling is necessary). Note that *SCFW* can perform only *readouts* of rectangular *ROIs*. Accordingly, if the active *request* describes a *ROI* with arbitrary shape, the ordinal number of each row in the *ROI* has to be regenerated, and loaded to *SCFW* via *ROI parameter FIFO interface* with the corresponding element available on *Xi FIFOs*. *Readout status FIFO* is filled with the proper status word at the end of the *ROI* descriptor transmission. The status contains the time of sampling and the start time of the last exposure before the sampling time among other things. The parameters of the active *request* available in *mode FIFO* are part of the readout status anyway as well.

Before the completion is indicated towards *Readout controller SM*, the busy signal of *SCFW* has to be examined: The number of 0-1-0 transitions during *READOUT* state have to be equal to the number of rectangular *ROI* descriptors which are related to the active *request* (see behaviour of busy in subsection 1.4.2). This condition ensures that the sampling triggered by the next *readout request* does not corrupt the image data of the actual *request*. The overwriting of image data is possible, because the download of *ROI* descriptors and the reception of the image data are overlapped in *SCFW*.

4.1.3 Command generator

Sensor module command generator (*SMCG*) provides various interfaces, which enables the other submodules of *SM* to easily initiate transmits of commands and to provide the data content of the appropriate commands. This subsection introduces the supported interfaces, the *FIFO* converters which transform the *FIFO* interfaces related to *readouts*, some submodules and the structure of the core which facilitates the unified transmission of every command.

Provided interfaces

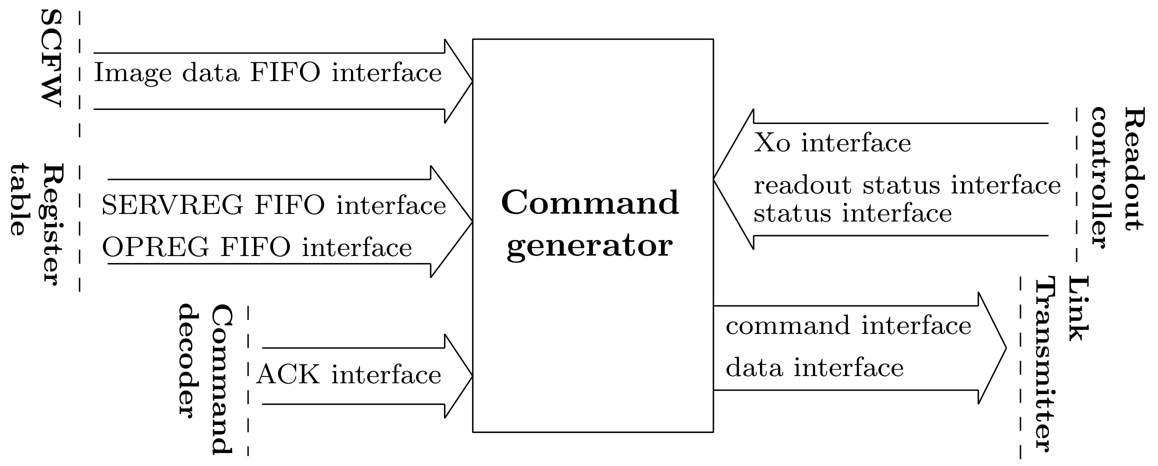

Figure 4.4. *Illustration of the interfaces provided by SMCG.*

The interfaces connected to *SMCG* are shown in figure 4.4. All of the interfaces detailed hereafter are used by *Command generator* to assemble complete commands which are transmitted via *Link Transmitter*. The ports related to the transmission of *ROI* data commands are *Xo*-, *Image data FIFO*- and *readout status interfaces* (FIFO interfaces). *Xo interface* grants the description of each row related to *readout requests*. If a *ROI* data command is sent towards *IPCU*, the row descriptors will be attached to the packet as well. *Readout status interface* indicates that a *request* is processed completely by *Readout controller*, and whether a *readout* was actually performed. The parameters of the *request* are also implied. These interfaces are described in detail in subsection 4.1.2. *Image data FIFO* interface is the image data port of *SCFW* which is introduced in subsection 1.4.2.

The other ports are *OPREG*, *SERVREG*, *status* and *ack interfaces*. The content of the *SM* register table can be quoted on *OPREG* and *SERVREG interfaces* before it can be sent to *IPCU*. *Ack interface* is used by *SM* command decoder, and facilitates the transmission of acknowledge commands with adjustable status information. The state of the *Exposure*- and *Readout controller* as well as the occupancy of the *mode FIFO* is accessible via *status interface*.

Readout FIFO converters

One of the most important tasks of *SMCG* is the assembling of *ROI* data commands which always contains *request* parameters (*ROIP* ID, *ROI* shape descriptor, etc.), *readout* status information (canceled, timeout) and optionally, image data. In order to apply the core unit structure also in *ROI* data commands, the already available interfaces have to be transformed. This modification is detailed here. The function of the converter modules is represented in figure 4.5.

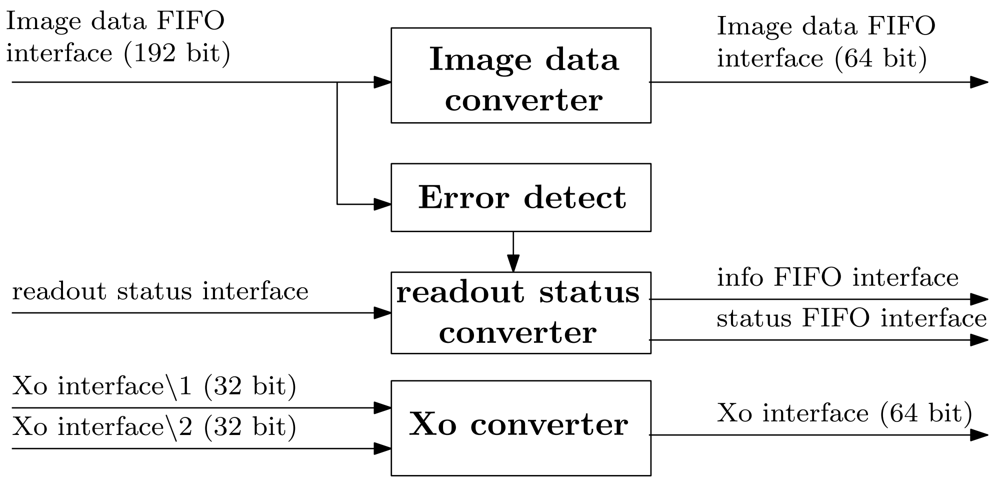

Figure 4.5. *FIFO converter modules in SMCG.*

Image data converter transforms the 192-bit image data port of *SCFW* into a 64-bit port. The data port with reduced width can be accepted directly by the *Core unit*. *Xo converter* produces a 64-bit *ROI* row descriptor interface based on *Xo interface*. This is necessary, because *Xo interface* consists of two 32-bit ports. The last elements related to a particular *request* are marked via a last bit in every interface. This rule is inherited by the regenerated ports as well, which is consistent with the operation of *Core unit*.

The derivation of *info*- and *status FIFO* is somewhat more complex. *Status FIFO* provides the length of *ROI* data command next to a bit which indicates whether image data is transmitted as well (*request* was not discarded). *Info FIFO* presents two kinds of information separated by the last bit: *request parameters* extracted from *readout status interface*, and an error word which indicates whether a timeout or cancellation occured during the *readout*. The timeout flag contains extra information compared to the timeout flag of *readout status interface*: the data stream received from *SCFW* provides an error bit which marks the invalid parts of the image data. Since this bit can transition to 1 even at the end of the data transmission, the error word can be attached to the end of the *ROI* data command (after the image data part). The observation of the image data stream is performed by *Error detect* which monitors the last, read and error signals of *Image data FIFO interface*. The error bit is connected to a sticky register (set by high input and separately can be reseted) which is examined at the arrival of the last signal.

Core unit structure

Core unit enables the state machine in *SMCG* to perform the transmission of every command uniformly. This description introduces the capabilities of the core rather than focusing on the particular realisation in *Command generator*.

This structure is based on some constraints: every commands are started with 1-3 *QWs* with arbitrary content followed by data which can be received via even 4 *FIFOs* sequentally. The *FIFOs* have to provide the same interface (data_o, empty_o, last_o and read_i). The last_o bit implements a control function, namely it indicates the boundaries of data segments.

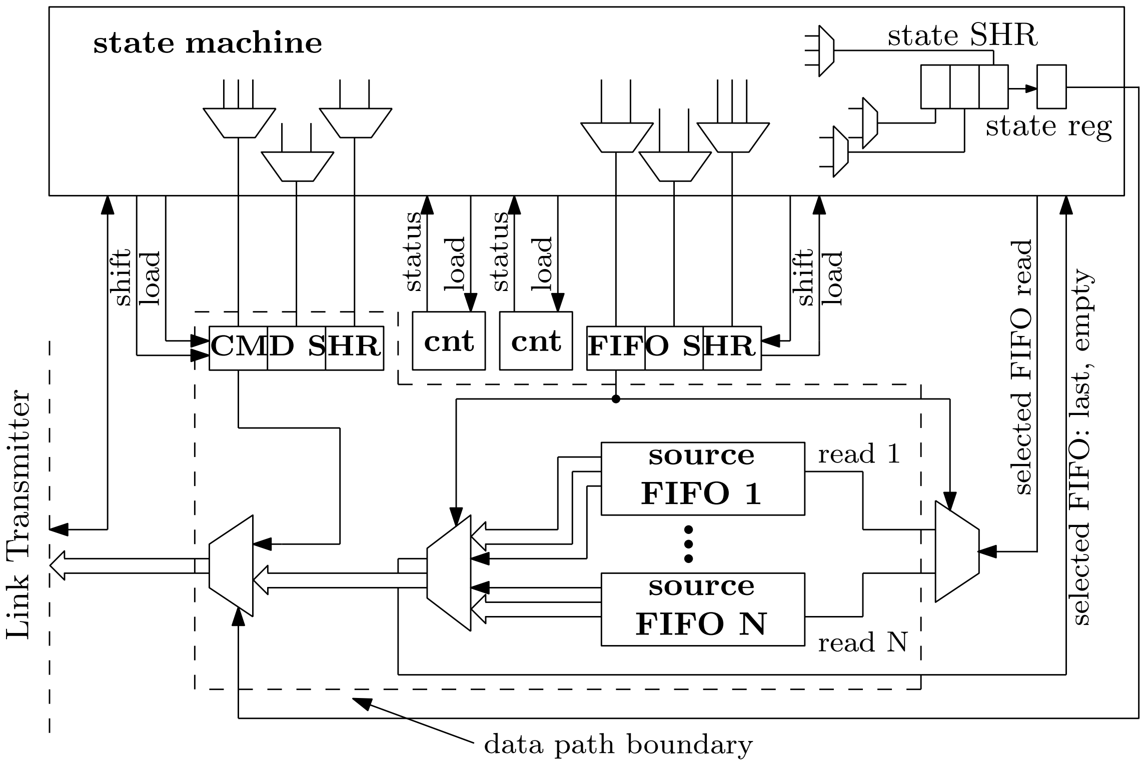

Figure 4.6. *Block diagram of SMCG core unit.*

The structure of the *Core unit* is shown in figure 4.6. which can be divided into a data path and a control path. The data path consists of the elements which are directly involved in the generation of the data stream. These elements are the *CMD SHR* (shift register), source *FIFOs* and the corresponding multiplexers. The data output can be driven by *CMD SHR* or by the data output of the selected source *FIFO*. This final routing is determined by the state of *state machine* (content of *state register*). The multiplexers and the encoder connected to the source *FIFOs* are controlled by *FIFO SHR*. Since *FIFO SHR* is a shift register, a predefined source *FIFO* sequence (during one command transmission) can be determined by the *state machine*. Both *FIFO*- and *CMD SHRs* are completed with a counter which contains the number of remaining elements in each *SHR*. To sum up, *FIFO* and *CMD SHRs* and the auxiliary counters allow the *state machine* to define the necessary phases in one step which realise the transmission of a command. The arrangement contains also a *state SHR* which defines the steps required by the current command. This module can accommodate the difference between commands, because arbitrary steps can be inserted during the transmission by adding extra states. The multiplexers and control signals (load, status, shift), used by the *state machine* to configure the sequence related to the selected command, are also tagged in figure 4.6.

The operation of *Core unit* is demonstrated with the description of a timing diagram (see figure 4.7.) which describes the steps and states performed during the transmission of a *ROI* data command.

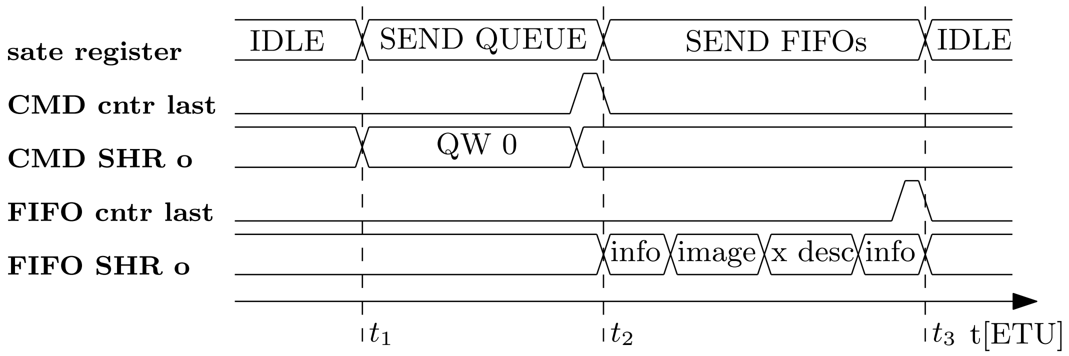

Figure 4.7. *Timing diagram of core operation during ROI data command.*

Status FIFO transisitons to not empty, so *state machine* launches a *ROI* data command at t_1. Since this command begins with only one arbitrary *QW*, this *QW* is loaded to *QW SHR* and *SHR cntr* is set to 1. The *ROI* data command is assembled from the contents of 4 *FIFOs*, so the *FIFO cntr* is set to 4 and *FIFO SHR* is loaded with the *FIFO* identifiers, which can be seen in the bottom row of figure 4.7. *CMD cntr last* signals by transitioning to 1 when all of the arbitrary *QWs* related to the command (currently 1) are sent (t_2). The contents of the *FIFOs* which are selected by *FIFO SHR* are transmitted in the following step: *readout request* parameters (*info*), image data (*image*), row descriptors (*x desc*) and error flags (*info* again). The assertion of the last bit related to the last *FIFO* in *FIFO SHR* (indicated by *FIFO cntr last* signal) determines the end of the transmission of the *ROI* data command (t_3). *State reg* transitions back to *IDLE* again, and is ready to send a new command.

4.2 IPCU related submodules

During my work, *ROI data processor* and *Event processor* were developed that are related to the *IPCU*. The details of these units are presented in the following. The simultaneous observation of figure 3.7. (*IPCU subsys* blockdiagram) is highly recommended in order to simplify the understanding of this section. The omitted modules are out of the scope of my work, and they are shortly introduced in subsection 3.2.2.

4.2.1 Event processor

The *event* handling mechanism of *EDICAM* is implemented by *Event processor*. The structure of this module is detailed in this part which implements the functionality introduced in subsection 2.3.2.

Event processor consists of *Input stages*, *Events* and *Action stage* as can be seen in figure 4.8. *Input stages* are responsible for the handling of *event* inputs, the *Event* blocks implement the binary operations on the corresponding inputs, and *Action stage* connects the *events* to the appropriate actions. Note that each *Event stage* provides more than one output toward *Action stage*, the so called *Aciton bits*.

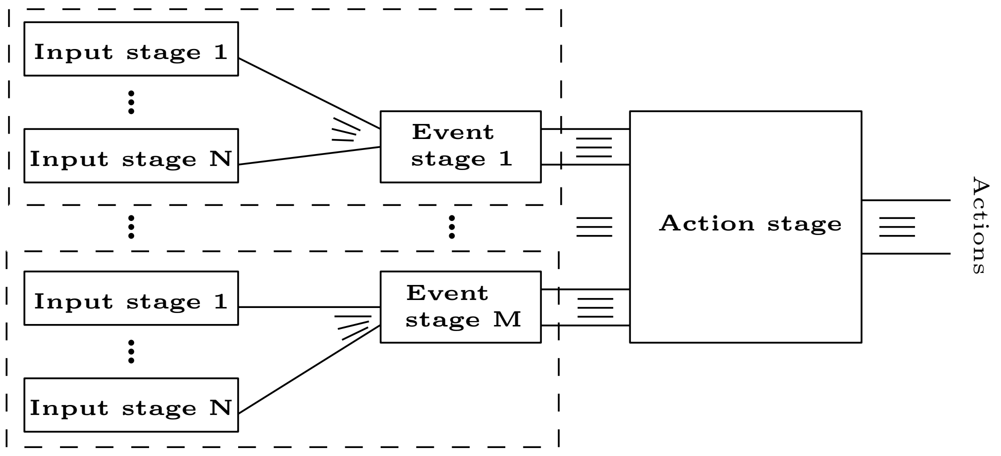

Figure 4.8. *Block diagram of Event processor.*

Input stage

The block diagram of *Input stage* can be seen in figure 4.9. Each *Input stage* consists of 4 input channels: *HOST*, *Image*, *External/evenet* and *ETU* time inputs. Only one of these channels can be connected to the output of the module by *Input type select* (see the multiplexer on the output). The operation of the channels is presented in the following.

The first channel is the so called *HOST* input which is driven directly by a register which is controlled by the *HOST* via *Register table*. The second channel is *Image* input which transitions to high level when an image parameter exceeds the threshold level (minimum, maximum or intensity), and the parameter is related to the *Reference ROI ID*. The *Image* input channel is sensitive only to one parameter which can be selected by *ROI mode select*. *Reference ROI ID* and *ROI mode select* are runtime configurable parameters. The last simple channel is *ETU time* input. This line is driven by a register which transitions to high level when the *ETU* time reaches an adjustable reference time.

External inputs and outputs of *events* are handled in the same channel (*External/event* input). One external input and one *event* output can be selected in the first stage (*External* and *Event* select). This part is followed by a multiplexer which determines whether the channel is sensitive to external inputs or *event* outputs. The remainder logic of this channel determines the handling of the selected input: *External/event* input can transition to high level or can be negated after a certain type of edge (rising or falling) occurs. The channel can also simply follow the state of the selected input (control mode). The select signals of the various multiplexers are runtime configurable.

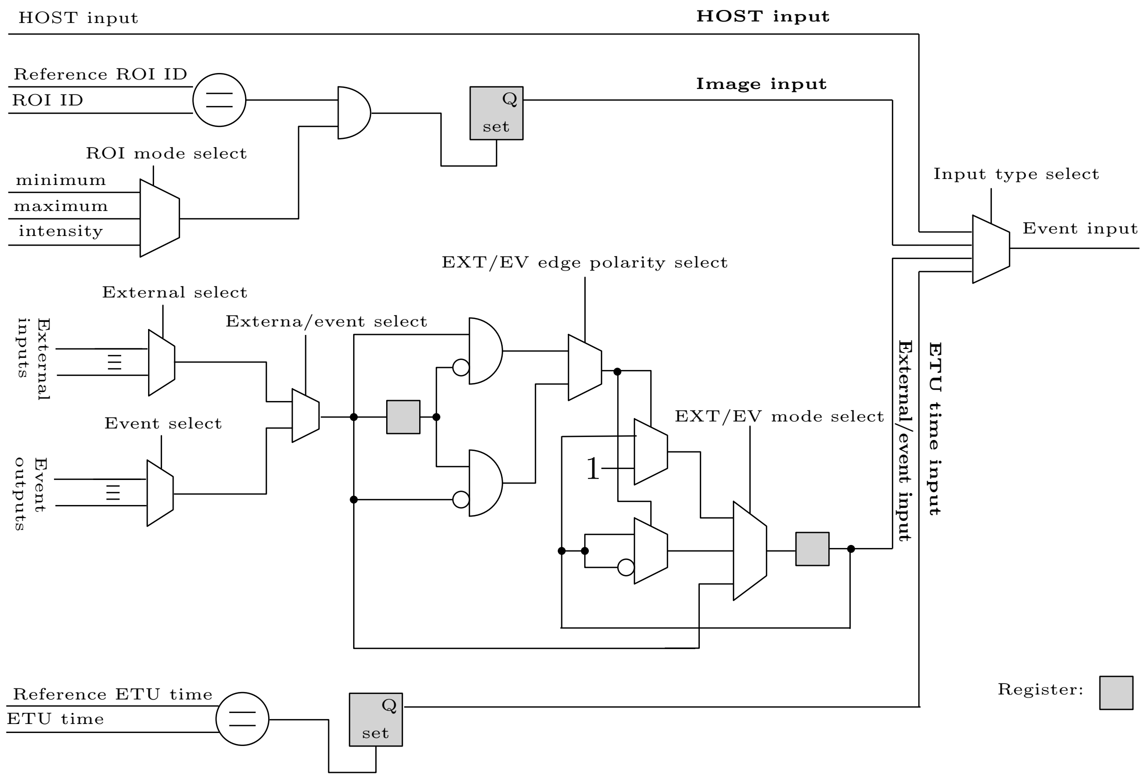

Figure 4.9. *Block diagram of Input stage.*

Event stage

The block diagram of *Event stage* is shown in figure 4.10.

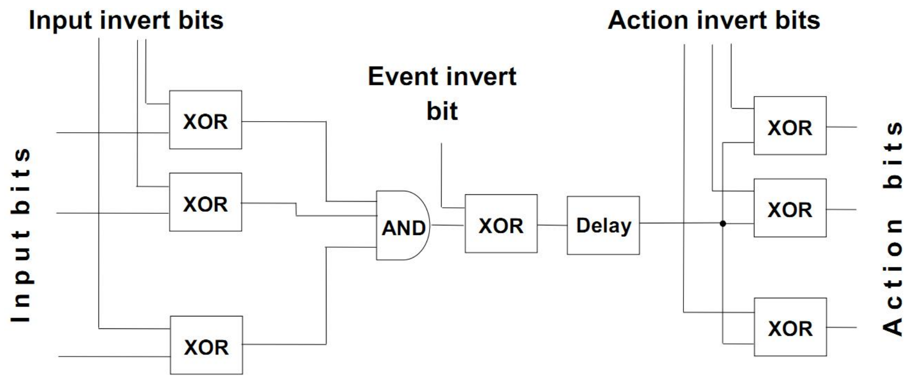

Figure 4.10. *Block diagram of Event stage.*

The *Input bits* are fed by the corresponding *Input stages*. After the operations described in subsection 2.3.2 have been performed, the generated *event* state is extended by the corresponding *Action invert bits*. The *Action bits* on the output are connected to the *Action stage*.

Action stage

Action stage is responsible for connecting the *Action bits* of the *Event stages* to the signals related to the predefined actions. The schematic representation of the module can be seen in figure 4.11.

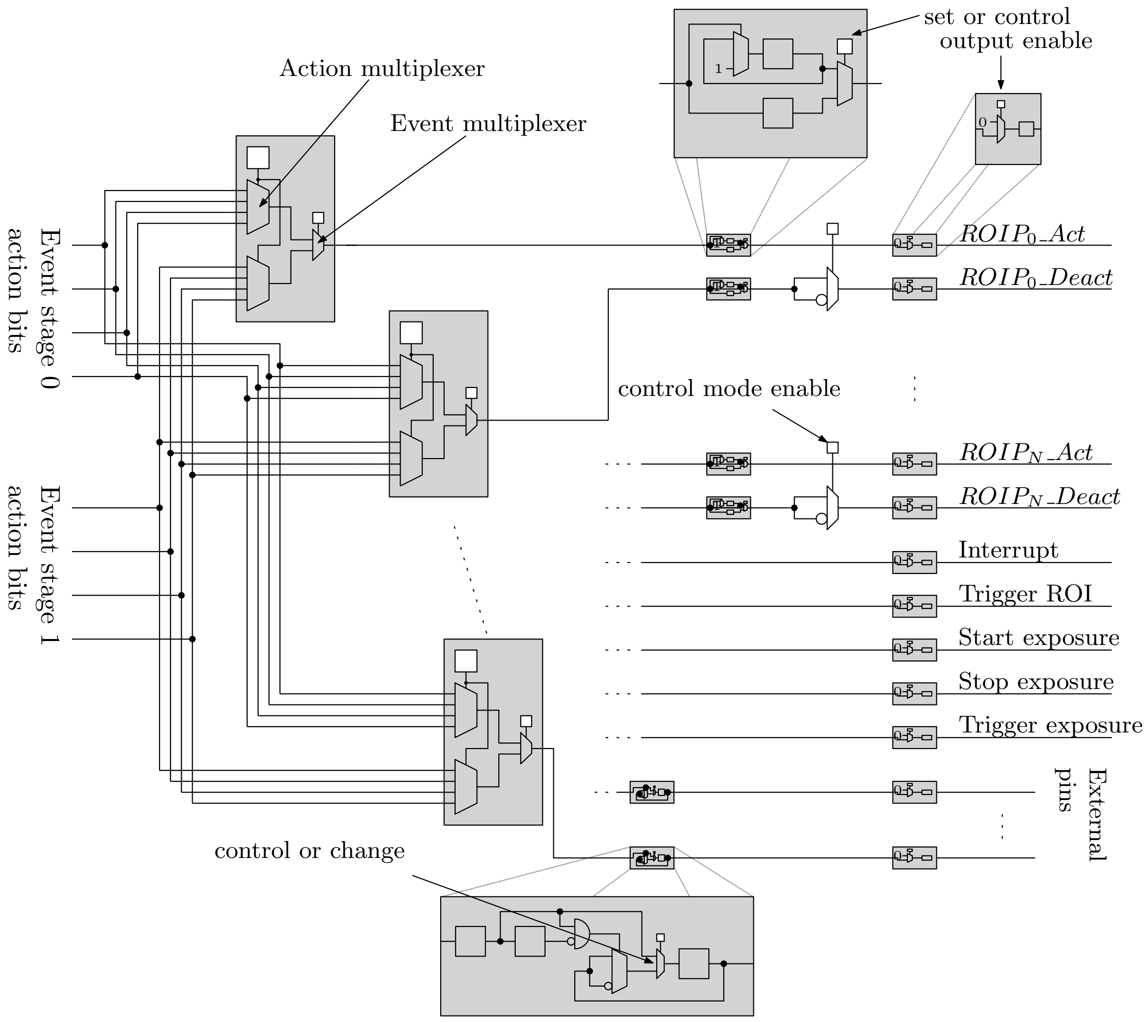

Figure 4.11. *Block diagram of Action stage (2 events and 4 actions/event).*

Each action can be connected to any input action bit via two kinds of multiplexers: The *Action* multiplexers select the same action bit of the *Event stages*, and the *Event* multiplexers differentiate between the *Event stages*. The outputs can also be disabled by *output enable*.

The *ROIP* and *External pin* related lines have adjustable sensitivity. The *Act* and *Deact* outputs can operate in a set mode (*set or control* and *control mode enable* is kept deasserted), which provides activation and deactivation of a *ROIP* only once (it remains unchanged afterwards). The other mode is control mode (*set or control* and *control mode enable* is kept asserted). In this case the same action bit controls simultaneously the *Act* and *Deact* signals of the same *ROIP* in a differential fashion.

The external pins have special sensitivity as well, because level control or state change on rising edge can be set via signal *Control or change*.

4.2.2 ROI data processor

EDICAM features image data processing as described in subsection 2.3.2. The corresponding functionality is implemented by *ROI data processor* (RDP). This module is fed by *Command decoder*, and drives some inputs of *Event processor*. The high level block diagram of *RDP* can be seen in figure 4.12. In this subsection, after the high level funcional operation of *RDP* is described, the submodules are introduced as well.

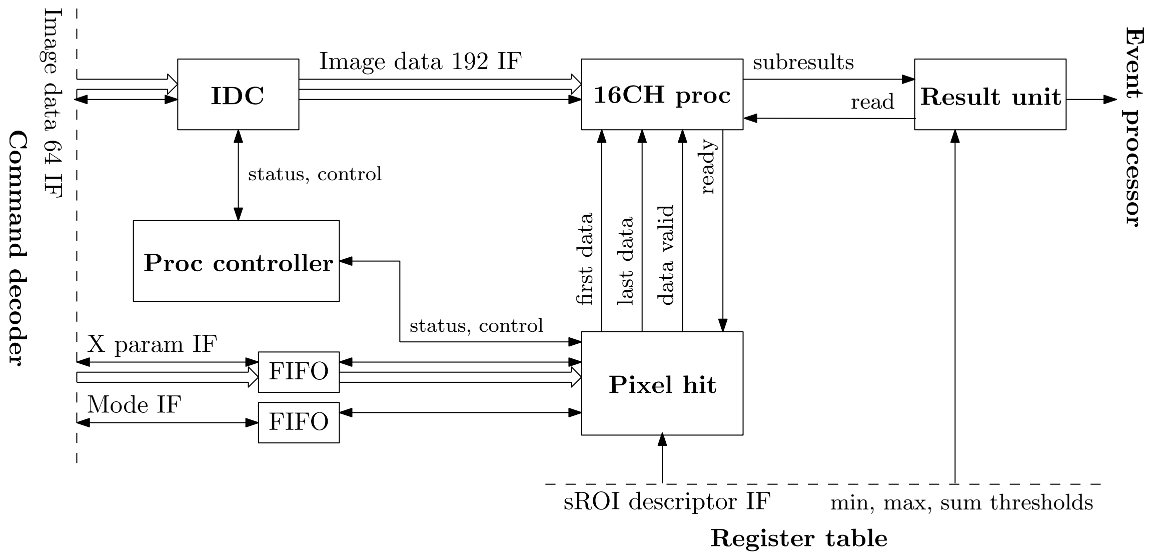

Figure 4.12. *High level block diagram of RDP.*

RDP receives image data, and calculates the minimum, maximum and sum of pixel intensity. The user can associate an *sROI* for each *ROIP ID* in the *Register table*. The relevant parameters are accessible via port *sROI descriptor IF*. Since image data is the result of a *readout* triggered by a particular *ROIP*, the data inherits the ID of the corresponding *ROIP* which is used to identify the corresponding *sROI*. The boundaries of the selected *sROI* limits the data involved in the processing: pixels outside the *sROI* are discarded.

The adequate *ROI* descriptor has to be provided also along with the image data, because of the *sROI* comparison. Image data and *ROI descriptors* which describe each row of the *ROI* are received via *Image data* 64 *IF* and *X param IF* respectively. These are the data interfaces of *RDP*, since the processing of information transmitted via these ports is controlled by *Mode IF*. *Mode IF* consists of parameters such as *ROIP ID*, *ROI shape* and vertical coordinate of the first row of the *ROI* among others.

Image data is processed by unit *16CH proc*, in blocks of 16 pixels as required by the *sROI*. The reproduction of coordinates and the comparison with the *sROI* boundaries are performed by unit *Pixel hit* (described later).

The width of the incoming image data is modified by *Input data converter* (IDC), which enables *16CH proc* to process 16 pixels simultaneously. The results of the *sROI* comparisons are also accessible on the output of *Pixel hit* in 16 bit wide segments.

The subresults of the channels are combined by *Result unit* after the current *ROI* is processed completely. The resultant minimum, maximum and sum parameters are compared with the threshold values related to the actual *ID*. These parameters are provided by *Register table* as well. The outputs of the comparators are routed to *Event processor*.

The synchronisation of the submodules is performed by *Proc controller*. This unit monitors the status of the *FIFO* outputs of *Input data converter* and *Pixel hit*, and lanches reads simultaneously. The informations transmitted by the two *FIFOs* are processed by 16*CH* proc parallel.

Input data converter

Image data is subjected to various width conversions during its lifecycle in *EDICAM*. These conversions are illustrated in figure 4.13., and are explained hereafter.

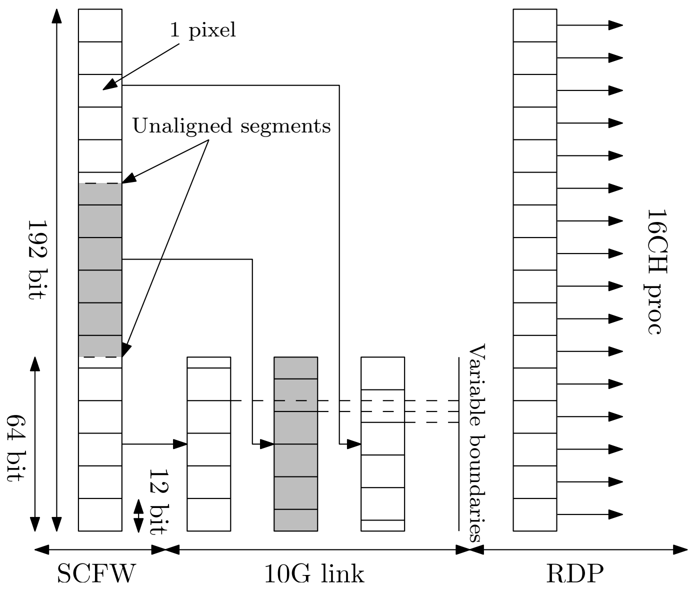

Figure 4.13. *Illustration of image data lifecycle.*

The image data is provided by *SCFW* via a 192-bit interface. Since the data port of 10*G link* has a width of 64-bit, the original segments have to be broken into 3 pieces before transmission. The image data is forwarded by *Command decoder* and by the intermediate modules directly, so the 12-bit pixel informations arrive at *RDP* in unaligned segments with variable boundaries.

In order to simplify the processing of the pixels, the unaligned property of the 64-bit wide data stream has to be terminated. This can be facilitated simply by converting back to the original 192-bit format. The block diagram of the implementation is shown in figure 4.14.

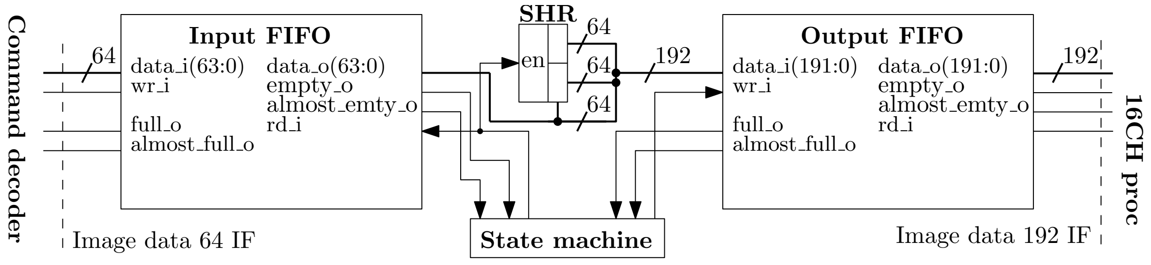

Figure 4.14. *Block diagram of Input data converter unit.*

The *Input FIFO* buffers the input data which is shifted into *SHR* (shift register) by *State machine*. If *SHR* is filled fully with two new *QWs*, and the third *QW* is available on the output of *Input FIFO*, *State machine* writes these three *QWs* into *Output FIFO* at once. The 192-bit wide image data is accessible on the output of this wider *FIFO* finally.

Image data 64 *IF* (see figure 4.14.) is driven by *Command decoder*, and the status and data signals of *Image data* 192 *IF* are connected to *Proc controller* and 16*CH proc* respectively.

Unit Pixel hit

This subsection introduces the block diagram of *Pixel hit*, and describes the submodules. *Pixel hit* consists of 3 main modules as can be seen in figure 4.15.

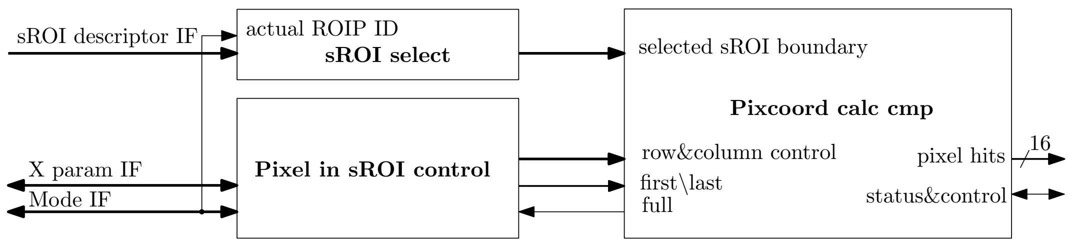

Figure 4.15. *Block diagram of Pixel hit unit.*

Pixcoord calc cmp unit calculates the coordinates of the pixels in the *ROI* related to the image data, and compares the values with the selected *sROI* boundaries. The results of the comparison are loaded into an output *FIFO*, which enables the driven modules to access the results of 16 comparisons at once via a simple *FIFO* interface. The selected *sROI* boundaries are provided by *sROI select unit.* This module compares the current *ROIP ID* with the IDs in the *sROI* descriptor part of the *IPCU* register table, and routes the boundary values related to the matching one to its output. *Pixcoord calc cmp unit* contains only the data path required by the calculation and comparison of coordinates which is controlled by *Pixel in sROI control unit.* This module controls *Pixcoord calc cmp unit* as required by the parameters accessed via ports *X param IF* and *Mode IF* (type and size parameters of the *ROI*).

The block diagram of *Pixcoord calc cmp unit* is illustrated in figure 4.16. which is divided into pipeline[1] stages. The description and operation of the stages are presented hereafter.

The values (row- and column number) stored in the base registers of *Stage* 1 can be reloaded, incremented or left unmodified. Signal *loadn_incr* changes between loading and incrementation, and takes effect if the corresponding *step* signal is asserted. The base register can be incremented by one at the top (row number), meanwhile one step increases the stored value with 16 at the bottom (column number). This difference is the consequence of the fact that the pixels in groups with a size of 16 are processed serially per each row. There has to be one break between two steps, because of the register inserted between the adder and the base register.

[1]Pipelining is a design technique which inserts registers between combinational logics to increase the maximum frequency.

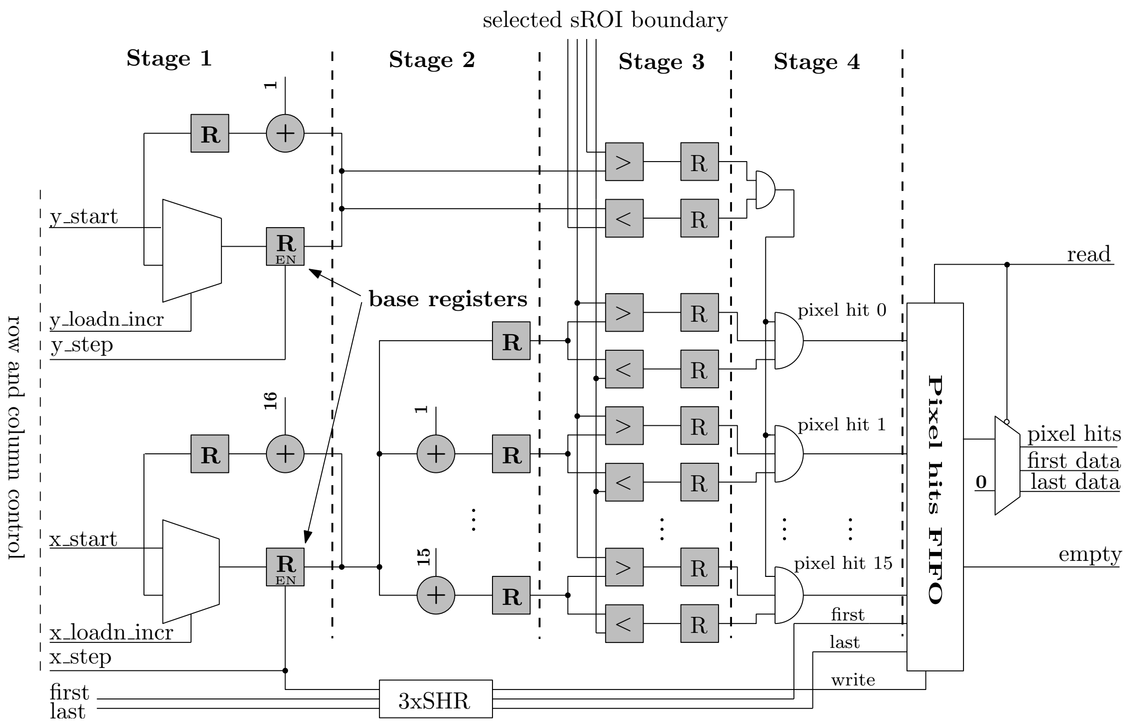

Figure 4.16. *Block diagram of Pixcoord calc cmp unit.*

Stage 2 produces the horizontal coordinates of 16 adjacent pixels at once which, along with the vertical coordinate of the row, are compared with the selected *sROI* boundaries in *Stage* 3. If both coordinates of a pixel falls into the *sROI*, the corresponding input bit of the output *FIFO* will be asserted by *Stage* 4. The generated bits are written to the *FIFO* by the delayed *step* signal. The other control signals (*last*, *first*) are delayed with the latency[2] of the path from the start input to the *FIFO* as well, which ensures the consistency of the stored results. The first and last groups of 16 pixels in the image data are designated by the *first* and *last* bits which will be used in 16*CH proc* for framing. Since the *FIFO* transfers control signals as well, its output is modified: If no read occurs, the output of the *FIFO* will be automatically deasserted. This behaviour is provided by the combination of a multiplexer and a register connected to the output of *Pixel hits FIFO*.

[2]Latency is the period required by the input to reach the output. Usually it is expressed in clock cycles.

Unit subROI select

The *sROI* boundaries are provided by unit *subROI select* which can be seen in figure 4.17. This module compares the *ROIP ID* of the image data with the *ROIP IDs* given in the *sROI* related part of the *Register table*. The comparison of IDs is performed by the logic at the bottom of figure 4.17.

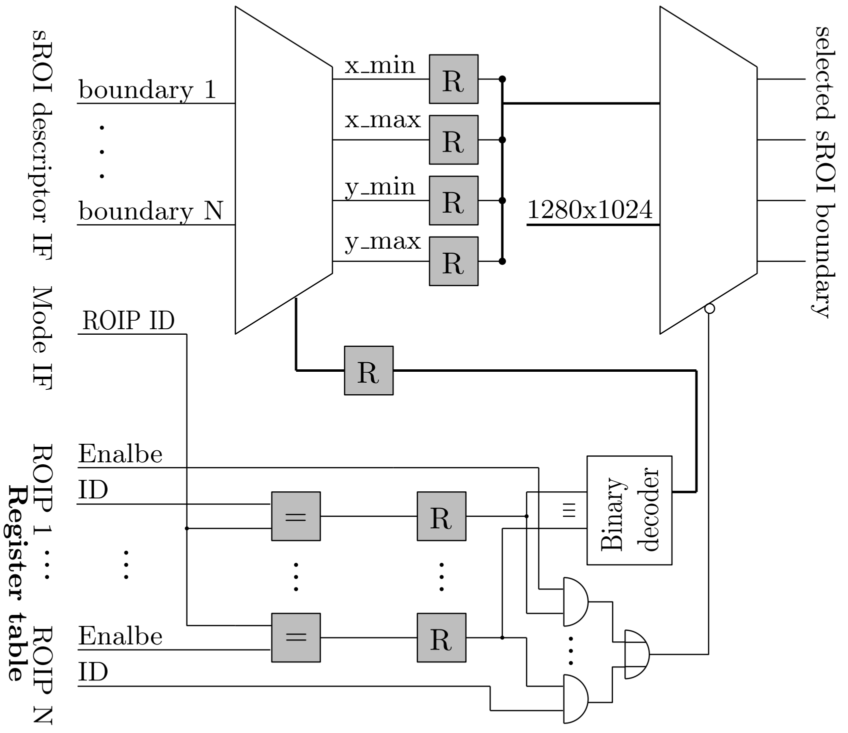

Figure 4.17. *Schematic representation of sROI select unit.*

Unless there is a match with a *ROIP ID* in the *sROI* table, the second multiplexer at the top will allow the processing of the full image. The output of the comparators are connected to a binary encoder as well which drives the select signal of the first multiplexer. The inputs of this multiplexer are fed by *sROI* boundaries related to the comparable *ROIP IDs*.

Unit 16CH proc

The 192-bit image data provided by *Input data converter* and the *sROI* comparisons generated by *Pixel hit* is processed by 16*CH proc*. This unit consists of 32 sequential comparators (left side in figure 4.18.) and 16 accumulators (right side in figure 4.18.). The comparators realise the minimum and maximum calculation, and the accumulators facilitate the derivation of the intensity. These submodules are organized into groups: two comparators and an accumulator is referred to as a *channel*.

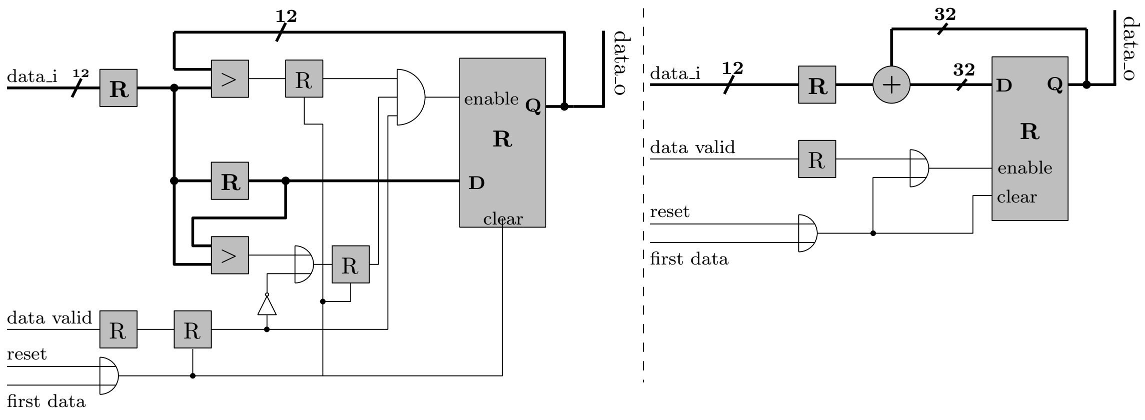

Figure 4.18. *Block diagram of the sequential comparator and accumulator.*

These submodules are highly pipelined. The corresponding registers are equipped with special reset signals driven by *Pixel hit* (*first data*), which supports the reception of consecutive image data packets. The results of comparisons between pixel coordinates and the selected *sROI* boundary are received via port *data valid*. This port enables or disables the update of the current result by the subsequent 12-bit element.

The flow control of 16*CH proc* during the processing of a complete image data packet is illustrated in figure 4.19.

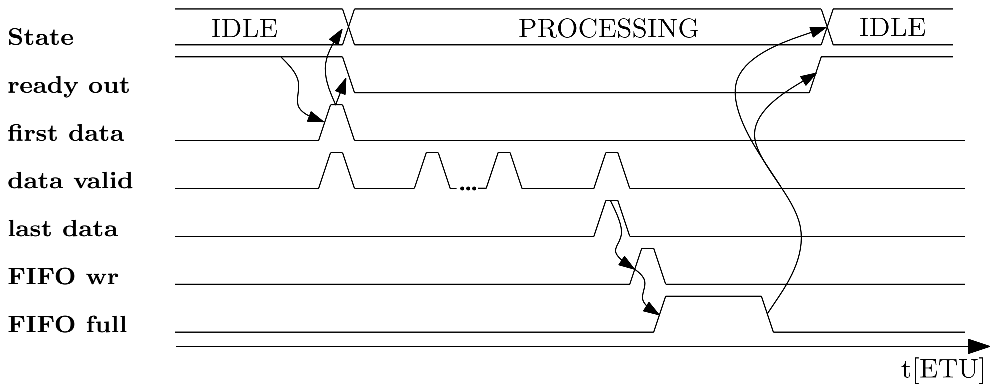

Figure 4.19. *Control flow of* 16*CH proc.*

A new data segment is processed when signal *data valid* is asserted, and the first and last segments of a *readout* are indicated via *first in* and *last in*.

The outputs of the *channels* are written to an output *FIFO* by signal *last in* (*Result unit* is fed by this *FIFO*). The *channels* can receive the data segments of the next *readout* after this output buffer has become not full (see *FIFO full* and *ready out* in figure 4.19.).

4.3 Common submodules

In this section modules which are instantiated in both *SM* and *IPCU* are introduced. These common modules are *Link Transmitter*, *Link Receiver* and *ETU timing unit* (measurement of time and interfacing the 10*G link* is mandatory on both sides).

4.3.1 Link Transmitter

Link Transmitter drives the *transmit interface* of 10*G link*. This module provides a *data-* and a *control interface* on its upstream ports[3], as shown in figure 4.20. The data to be transmitted can be written on the *data interface* continuously which is realised as a simple *FIFO* interface. This raw datastream is cutted into packages and forwarded by *Link Transmitter* as it is described by the *control interface*. Only the size of the required package has to be defined: *Link Transmitter* asserts the *start-* and *end of packet* signals of 10*G link* in the appropriate *QW* positions. *Transmitter* generates the *CRC* codes of packets which are attached to the end of the corresponding packets. The control signals of *control interface* (*request* and *acknowledge*) are implementing a simple handshaking protocol.

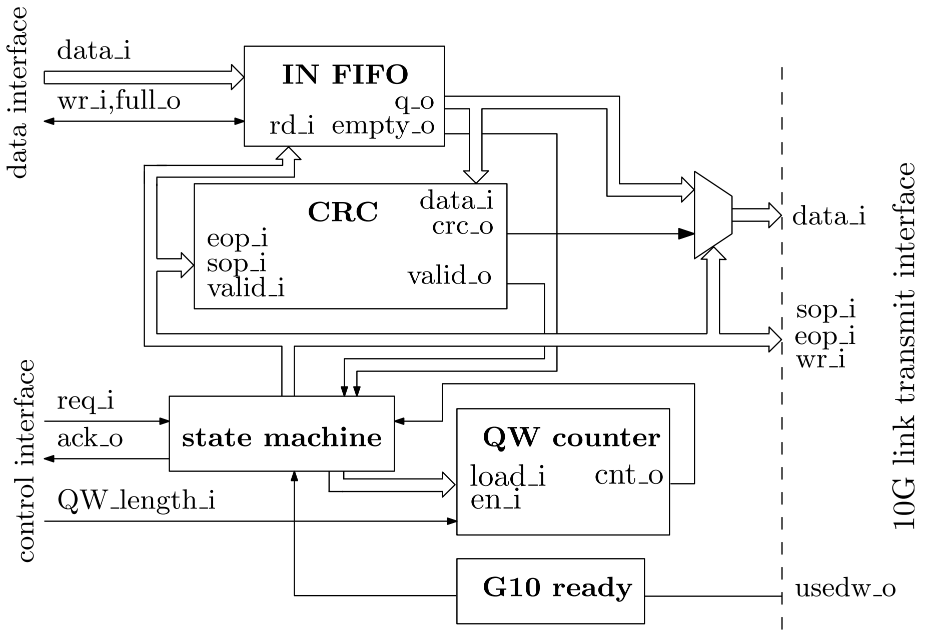

Figure 4.20. *Block diagram of Link Transmitter.*

The block diagram of *Link Transmitter* can be seen in figure 4.20. The control of *CRC* and the read port of *IN FIFO* is performed by *state machine*. This module is completed with *QW counter* which is used to track the state of the transmission of a packet. *G*10 *ready* monitors continuously the *usedw_o* port of 10*G link*, and indicates whether the data input *FIFO* of 10*G link* is full. If *IN FIFO* is not empty, 10*G link* is not full and a packet sending is in progress, *state machine* controls the read, write,

[3]An interface of a module can be referred to as upstream port if it communicates with a higher level layer relative to the module.

start and end signals of the units. At the end of every packet, the multiplexer which drives the data input of 10*G link* is switched to the output of the *CRC* module. When also the *CRC* code is transmitted with signal *end of packet* kept high, the next request on *control interface* can be acknowledged.

4.3.2 Link Receiver

Link Receiver receives packets sent by *Link Transmitter* via the 10*G link*, and performs length and *CRC* check. The block diagram of the unit is represented in figure 4.21. The examination of the packet length requires the size to be implied in the data stream explicitly next to the start- and end of packet signals. Since the first *QW* of every command of *EDICAM* contains the size of the command, this criterion is self-satisfied. The *CRC* is regenerated and compared with the received one. The interface and operation of this module are introduced hereafter.

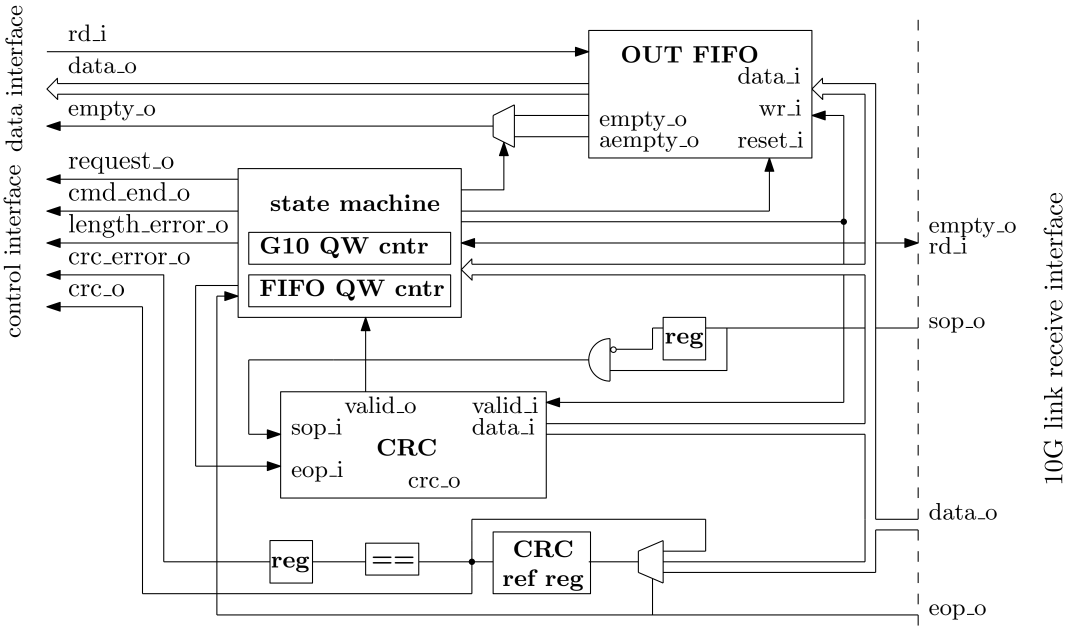

Figure 4.21. *Block diagram of Link Receiver.*

The results of the processing of the received packets performed by *Link Receiver* are presented on two upstream user interfaces, namely on *control interface* and *data interface*. The received *QWs* are accessible one-by-one via *data interface* which is implemented as a simple *FIFO* interface. The logic connected to *Link Receiver* has to moninor continuously the *control interface* during reading the *data interface*. The arrival of a new packet is indicated on *request_o* which transitions back to zero after a read occurs on *data interface*, or if a reception is aborted internally due to errors. Unless there is a length error, *cmd_end_o* is asserted for one period after the last *QW* of the currently received packet is read via *data interface*. In this case, *crc_error_o* is also kept low along with *length_error_o* if the bits are received properly. The transmitted *CRC* is available on *crc_o* while *cmd_end_o* is active. In contrast, if there is a length error, *cmd_end_o* will be asserted immediately after the deviation is detected, with all of the error signals set to high level. The logic connected to *data interface* has to discard all the *QWs* which have been received since the last *request_o*.

To complete the above presented description of the ports of *Link Receiver*, some internal detailes are described in the following. The control module of *Receiver* is referred simply to as *state machine*. This unit contains two counters to track the state of the transmission of a packet. *G10 QW cntr* is used to store the number of the *QWs* which have not been received from 10*G link receive interface* yet, and are related to the currently received packet. The other counter, namely *FIFO QW cntr*, contains the number of *QWs* of the current packet which are not read from *OUT FIFO* yet. *FIFO cntr* enables *state machine* to determine the time, when *cmd_end_o* has to be asserted. *G10 cntr* supports the check of the length of packets. Besides the immediate assertion of the error and end signals when a length error occurs, the *OUT FIFO* is reseted (there can be *QWs* of only one packet in *OUT FIFO* at the same time). If the packet is longer than expected, the remaining *QWs* will be automatically flushed from the *receive interface* until a *QW* is read with the end of packet signal kept high.

The last important element of *Link Receiver* is the multiplexer connected to the empty outputs of *OUT FIFO*. Since the almost empty status signal of *OUT FIFO* is kept high when only the last *QW* of the current packet has not been read yet, this component can be used by *state machine* to block the transmission of the last *QW* until the corresponding *CRC* is calculated. This enables *control interface* to produce the end and error signals parallel with the last *QW* of a packet, which simplifies the upstream logic.

4.3.3 ETU timing unit

The current version of *ETU timing unit* can produce the system time with 100 ns resolution. The source clock is permitted to have a frequency of any multiple of 10 MHz, and can be unrelated to the system clock. The input clock period has to be known compile time, and must not change during operation. The structure (see figure 4.22.) and operation of this module is introduced in the following.

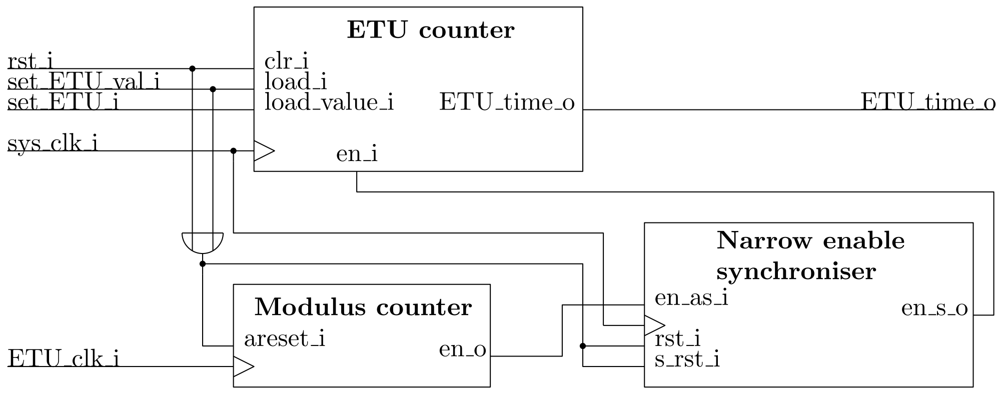

Figure 4.22. *High level block diagram of ETU timing unit.*

Basically, *ETU* time is produced by a binary counter called *ETU counter* which is an instance of the *lpm_ counter* megafunction[4]. The clear and load ports of the counter are connected to the ports of *ETU* timing unit, which enables both the firmware and user to clear or set the system time.

Since *ETU* counter is controlled by the system clock (*sys_ clk_ i*) whose period presumably differs from 100 ns, its enable input is used as well. The enable input is driven by the *Modulus counter* (another *lpm_ counter* instance completed with auxiliary logic) which performs the division of the *ETU* source clock (*ETU_ clk_ i*). The divider is fixed, and the signal produced by *Modulus counter* (*en_ o*) contains pulses periodically with a length of *ETU* source clock.

The enable signal generated by *Modulus counter* (*en_ o*) is asynchronous to the system clock, so it has to be synchronised. This function is performed by *Narrow enable synchroniser* which can process both wide and narrow input pulses, and produces pulses on its output (*en_ s_ o*) synchronous to the target clock domain with a length of the destination clock period.

Both the reset and load of *ETU counter* clears *Modulus counter* and *Narrow enable synchroniser* as well, which ensures the new *ETU* time to be stable for one *ETU* after this events.

Narrow enable synchroniser

This module provides robust synchronisation of enable signals, because both wide and narrow enable signals can be properly received by the applied structure. The schematic representation of the design can be seen in figure 4.23.

If an enable signal arrives from a faster clock domain, the receiver register controlled by the slower clock will be likely to miss the narrow pulse. By its nature, this problem cannot be resolved by any synchronous design method. To receive such a signal, the input pulse has to be stretched first. This is performed by the *stretcher D FF* whose clock input is fed by the enable pulse. Since the data input of the *stretcher* is connected to '1', the output of the strether transitions to high level along with the enable pulse.

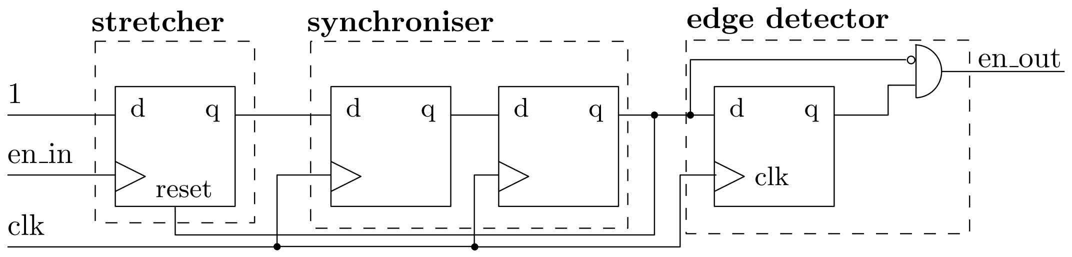

Figure 4.23. *Block diagram of the narrow enable synchroniser.*

The *stretcher* drives a synchroniser chain which produces an impulse with a length of more clock periods in the receiver domain. Finally, the second synchroniser register clears the *stretcher* via its asynchronous reset port, so the synchroniser registers are also cleared sequentally. The introduced modules have to be completed with an edge detecor, because

[4]Megafunctions are parameterizable pre-designed modules provided by *Altera*.

the output of the synchroniser is asserted longer than one clock period in the receiver domain.

Since not the duration but the rising edge of the received enable signal influences the operation, wide enable pulses can be accepted by this arrangement as well. The edge sensitivity comprises also a restriction, namely the incoming enable pulse must be glitch free, because the shortest transients will be caught too.

The introduced module does not require any constraints in the corresponding *SDC* file, because the clock of *stretcher* and the first synchroniser *D FF* are automatically unrelated (the incoming enable signal is not a clock presumably), so the path feeding the reset port and the path fed by the data output of *stretcher* are considered false paths. The other paths of this design are analysed normally (as a synchronous circuit).

Chapter 5

Verification of the firmware

This chapter introduces a basic test scenario. The results are based on the observation of the programmed hardware (7th step of the design flow introduced in section 1.5). The control of the firmware was performed by an application running on the *HOST* computer. The applied logic analyzer is introduced shortly as well.

5.1 SignalTap II Logic Analyzer

Altera supports design debugging by providing the *SignalTap II Logic Analyzer*. This logic analyzer allows the examination of the internal signals without using any extra *IO* pins, while the design is running at full speed. The analyzer is made of general *FPGA* logic resources, however, its utilization is negligible compared to the capacities of the applied *FPGAs*.

The signals in the test are controlled and sampled by the same clock signal in a single diagram. Transitional storage qualification was applied: sampling was performed only after the value of certain signals had been changed. This feature allows the observation of periods which are very long relative to the period of the sampling clock. The impact of this feature on the waveforms has to be considered during the interpretation of the diagrams presented hereafter.

5.2 Demonstration of a readout scenario

The basic features of *EDICAM* are demonstrated in this section: *exposure* and *readout* operations as well as the *SM-IPCU* communication from a high level point of view are exhibited. These operations are introduced in subsection 2.4.1, 2.4.2 and 2.5.

The transmitted commands and the interfaces of *SCFW* are shown in the test results. The interpretation of *SCFW* ports is presented in subsection 1.4.2.

Scenario description

The firmware performs two *exposures* and one *readout*. The parameters of the exposure sequence are the following:

- $t_0 = 29000000\ ETU(2900\ ms)$
- $T_{exposure} = 2000000\ ETU(200\ ms)$
- $T_{repetition} = 10000000\ ETU(1\ s)$
- $N_{loop} = 2$

One *ROIP* is activated with a t_{start} of $40000000\ ETU(4\ s)$. The *ROIP* has rectangular shape (1 pixel height and 32 pixel width). The upper left corner of this rectangle and the full image is in the same position.

IPCU-SM interaction

The commands received by *SM* and *IPCU Command decoder* are shown in figure 5.1. and 5.2. respectively. Since *SM* acknowledges every command, and reports the state transitions of the *exposure* and *readout* related submodules, the acknowledged commands and the received *SM* states can be seen in figure 5.2. as well. The timescales of the diagrams are in chronological order.

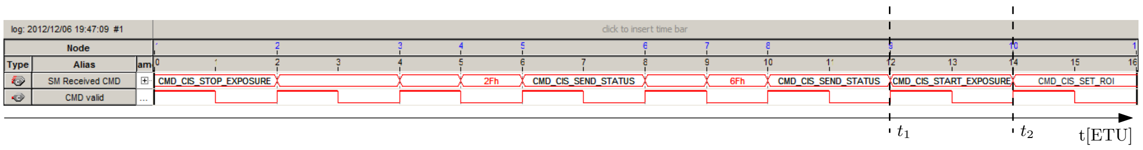

Figure 5.1. *Commands received by SM Command decoder.*

SM receives *CIS_START_EXPOSURE* command at t_1, which asserts the exposure control signal of *SCFW* at $39000000\ ETU$ ($3.9\ s$). This command is followed by *CIS_SET_ROI* at t_2 which defines the parameters of the *readout request*.

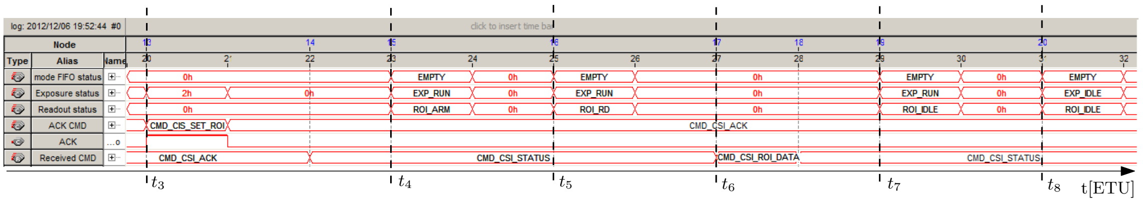

Figure 5.2. *Commands received by IPCU Command decoder.*

The events triggered by the *readout request* (t_2) can be seen in figure 5.2. First, *CIS_SET_ROI* is acknowledged at t_3. At t_4, *Readout status* changes to *ARMED* (*ROI_ARM*) after the *readout request* has been loaded to *Redout controller*. The sampling time of the *request* arrives at t_5, and *Readout status* changes to *READOUT* (*ROI_RD*). A command which contains image data (*CMD_CIS_ROI_DATA*) is received at t_6 due to the *readout*. The status changes of *Exposure-* and *Radout controller* are received at t_7 and t_8 (*IDLE*): default states arise after the *exposures* and *readouts* have been completed.

Low level operation of SM

SCFW ports as well as the system-time and the states of *Exposure*- and *Readout controller* are shown in figure 5.3. This signal composition represents the low level operation of the firmware during the *readout*. The interpretation of the signal transitions is presented in the following.

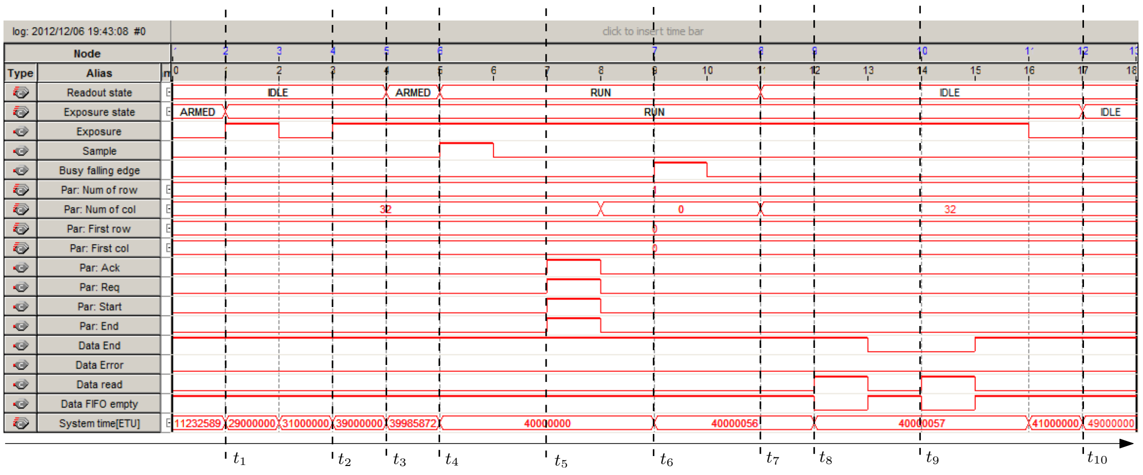

Figure 5.3. *SCFW interfaces and SM subsys states during a readout.*

Exposure state changes from *ARMED* to *RUN* at t_1: the start time of the first exposure arrives (29000000 *ETU* (2.9 *s*)). The exposure control signal (*Exposure*) is deasserted at 31000000 *ETU* (3.1 *s*), and the next *exposition* begins at t_2 (39000000 *ETU* (3.9 *s*)).

The *readout request* is received by *SM* at t_3, and hence *Redout state* changes to *ARMED*. *Redout controller* is waiting for the *sampling time* beyond t_3, the system-time reaches 40000000 *ETU* (4 *s*) at t_4, so *Sample* is asserted for one period. Note that *Exposure* has been asserted for 1000000 *ETU* (100 *ms*) at this moment.

Sampling is followed by the download of the *ROI* parameters via *ROI parameter FIFO IF* at t_5. The values of signals prefixed with *Par* in the diagram comply with the parameters of the *ROI* (0,0 start position, 1 height, 32 width). *SCFW* allows the launch of the next sampling by the deassertion of signal *busy* (t_6) after the image data has been transmitted completely by the *ADCs*.

Since *Readout controller* is responsible only for the control of the sampling and *ROI* parameter related signals of *SCFW*, it steps into *IDLE* state at t_7 (*busy* has to be deasserted as well).

The *Image data FIFO IF* of *SCFW* is connected to (*SM*) *Command decoder*, and the transmission of the image data between *SCFW* and *Command decoder* is performed at t_8 and t_9 (see signals prefixed with *Data*). Note that 32 pixels generate two 192-bit data segments, and *Data end* is asserted after the last reading via *Image Data FIFO IF*.

Sampling and data transmission is performed quickly after 30000000 *ETU* (3 *s*), and hence *Exposure controller* steps into *IDLE* just at t_{10}. The complete *SM* changes to its default state after the commands sent by the *ICPU* have been executed.

Chapter 6

Summary, outlook

In summary, it can be said that the inital objectives have been completely achieved. The implementation introduced in this thesis complies with the high level specification of *EDICAM*: the integrated system is capable to perform highly customizable image acquisition. I appreciate mostly the skills and experiences acquired during the work next to the success of the workmanship. This chapter introduces these capabilities, the corresponding tasks and the development opportunities of the current implementation.

6.1 Design experience

Since the development of *EDICAM* is practically digital design, my basic craftmanship related to this speciality had to be improved at the very start: *RTL* design must not be started without professional expertise.

The inital theoretical knowledge was strengthened during the design of simpler submodules. These units were verified via small simulations which laid down the basics of my verification skills. The initial submodule implementations and simulations were followed by increasingly higher level submodule integrations and simulations. The independent and extensive comprehension of the specification become essential at this stage.

The complete understanding of the integrated firmware and the continuous expansion of my digital design knowledge facilitated the revision and review of the system level considerations. The modifications resulted in speed and area improvements, which allow the further expansion of the firmware (without any limitations caused by the currently available hardware resources).

System level simulations and tests were performed after the firmware had been integrated. This part of the design flow requires the ability to apply effectively the various debug and test tools. The troubleshooting in such a large system has an impact on the outlook of the designer: the importance of the appropriate hierarchy and simulation were unraveled.

The above introduced progress is practically the way which is required to understand the complex design flow of *FPGA* development. This capability is essential in optimal *FPGA* design. *I thank ProDSP Ltd. effusively the opportunity to participate in the design of such a challenging complex system.*

6.2 Further improvements

The introduced version of *EDICAM* has numerous improvements compared to its previous version, however, there are plenty of development opportunities.

The introduced *ETU timing* unit can be further improved. The next version of the module will support external clocks received via the extension card. The frequency of this external clock does not have to be specified in compile time (it still has to be in a certain interval), and hence the multiply and divide parameters of the corresponding *PLL* have to be adjustable in runtime. The frequency of the *ETU* clock which is derived from an external source has to be automatically verifiable.

Another weakness of the system is related to the storage of *ROI* descriptors. These parameters are stored currently in dedicated *RAMs*. These *RAMs* have predefined number and size. This realization is suboptimal: each (even a small) *ROI* occupies a complete dedicated *RAM* unnecessarily, because the architecture allows the storage of only one *ROI* descriptor in a *RAM*. The *ROI* descriptors could be stored in a common memory (with increased depth) in a following version. This approach facilitates the improvement of the usability of the area intended to *ROI* descriptor storage, but raises issues related to the management of the placement of the descriptors.

The boundary of a particular *ROIP* is defined, and cannot be modified after it has been downloaded to the firmware. The so called *moving ROIPs* are intended to resolve this constraint: The boundary of such a *ROIP* changes in runtime without any intervention. The course is specified by predefined attributes.

The last improvement is related to the *exposure* mechanism. The derivation of the pixel-by-pixel difference between images related to two consecutive *readouts* is required in some applications. Since the current exposure mechanism allows the acquisition only of the raw output of the sensor, two independent *readouts* have to be performed first. The intensities related to the second *readout* have to be extracted from the first by the application running on the *CPU*. This mechanism will be directly supported by the so called *differental ROIPs* in a future version. Since the storage of a complete image is required to implement this funcionality in the firmware, *DDR* memories (located in the *Altera board*) have to be applied.

Bibliography

[1] Stellarator. `http://en.wikipedia.org/wiki/Stellarator`, October 2012. Wikipedia article.

[2] Tokamak. `http://en.wikipedia.org/wiki/Tokamak`, November 2012. Wikipedia article.

[3] Altera Corporation. *Stratix II Performance and Logic Efficiency Analysis*, September 2006. Document version: 2.0.

[4] Altera Corporation. *Stratix II GX Device Handbook, Volume 1*, October 2007. Document version: 2.2.

[5] Altera Corporation. *Stratix II GX PCI Express Development Board Reference Manual*, March 2008. Document version: 1.0.1.

[6] Altera Corporation. *PCI Express Compiler User Guide*, March 2009. Document version: 9.0.

[7] PONG P. CHU. *RTL HARDWARE DESIGN USING VHDL*. John Wiley and Sons, Inc., 2006.

[8] Cypress Semiconductor Corporation. *LUPA-1300 Datasheet*, December 2004.

[9] Endymion. Kernfusion: Wendelstein 7-x erreicht ersten meilenstein. *Natur & Umwelt*, March 2008. `http://blog.safog.com/2008/03/10/kernfusion-wendelstein-7-x-erreicht-ersten-meilenstein/`.

[10] Steve Kils. *Advanced FPGA Design Architecture, Implementation, and Optimization*. John Wiley and Sons, Inc., 2007.

[11] Ryan Scoville. *TimeQuest User Guide*, December 2010. Wiki Release 1.1.

[12] Xilinx. *Spartan-3E FPGA Family: Data Sheet*, August 2009. Document version: 3.8.

Appendix

A.1 Abbreviations

ADC	Analog Digital Converter
ALM	Adaptive Logic Module
CMD	Command
CRC	Cyclic Redundancy Check
DMA	Direct Memory Access
EDICAM	Event Detection Intelligent Camera
EOP	End Of Packet
ETU	EDICAM Timing Unit
FF	Flip-Flip
FIFO	First In First Out
FPGA	Field Programmable Gate Array
Gbps	Gigabits per second
HDL	Hardware Description Language
IF	Interface
IPCU	Image Processing and Control Unit
LSB	Least Significant Bit
MTA W FK	Magyar Tudományos Akadémia Wigner Fizikai Kutatóközpont
uTco	T_{cq} in TimeQuest
uTh	T_h in TimeQuest
uTsu	T_s in TimeQuest
PCIe	Peripheral Component Interconnect express (PCI express)
PLL	Phase Locked Loop
RCG	Readout Command Generator
RDP	ROI data processor
ROI	Region Of Interest
ROIP	ROI Process
RTL	Register Transfer Level
SCFW	Sensor Conrol Firmware
SHR	Shift Register
SM	Sensor Module
SMCG	Sensor Module Command Generator

SMCQ	Sensor Module Command Queue
SOP	Start Of Packet
sROI	sub-ROI
T_{cq}	Clock to q delay
T_s	Setup time
T_h	Hold time
QW	Quadword