FPGA-based Camera Readout for the Mu3e Experiment

Johannes Gutenberg Universität Mainz

Prof. Dr. Niklaus Berger, Prof. Dr. Bertil Schmidt

Bachelor's Thesis

Philipp Benjamin Vitus Freundlieb

submitted March 3, 2023

Abstract

The Mu3e Experiment is searching for lepton flavour violating decay of muons, by firing high rates of them against a stopping target inside a custom detector assembly, measuring the resulting decay particles. To improve the accuracy of the particles' paths' reconstruction, by measuring the exact sensor positions, a camera alignment system has been proposed.
This work explores the protocols associated with reading out CSI-2-based image sensors and proposes an FPGA-based prototype implementation of a CSI-2 bridge system, before weighing its applicability, considering said alignment systems requirements.

Kurzfassung

Das Mu3e-Experiment sucht nach einem Leptonen-Flavour-verletzenden Zerfall von Myonen, indem es hohe Raten eben dieser auf ein Target innerhalb einer eigens konstruierten Detektorvorrichtung schießt und die resultierenden Zerfallsteilchen misst. Um die Genauigkeit der Teilchenbahnrekonstruktion zu verbessern, wurde ein kamerabasiertes Ausrichtungssystem vorgestellt.
Diese Arbeit untersucht die Protokolle, die zum Auslesen von CSI-2-basierten Bildsensoren verwendet werden, und schlägt eine FPGA-basierte Prototyp-Implementierung eines CSI-2-Bridge-Systems vor, und prüft dessen Anwendbarkeit unter Berücksichtigung der Anforderungen an das genannte Ausrichtungssystem.

Declaration of Authorship

I hereby declare that I have written the present thesis independently and without use of other than the indicated means. I also declare that to the best of my knowledge all passages taken from published and unpublished sources have been referenced. The paper has not been submitted for evaluation to any other examining authority nor has it been published in any form whatsoever.

Mainz, March 3rd 2023

FPGA-based Camera Readout for the Mu3e Experiment
Johannes Gutenberg Universitaet Mainz

Acronyms

ARM Advanced RISC Machines (processor architecture)

ASIC application specific integrated circuit

BSM (physics) beyond the standard model

CCI camera control interface

CIL control and interface logic

CMOS complementary metal-oxide semiconductor

CRC cyclic redundancy check

CSI-2 camera serial interface 2

CSV comma separated values

DAQ data acquisition system

DDR double data rate

DI data identifier

DT data type

ECC error correction code

EMI electro magnetic interference

FEB front-end boards

FIFO first in first out

FPGA field programmable gate array

FSM finite state machine

GPU graphics processing unit

HS high speed

HV-MAPS High-Voltage Monolithic Active Pixel Sensors

I3C improved inter-integrated circuit

IP intellectual property

ISP image signal processor

I^{2}C inter-integrated circuit

LED light emitting diode

LP low power

LPDT low power data transmission

LVDS low voltage differential signalling

MCU micro controller unit

MIDAS Maximum Integrated Data Acquisition System

MIPI mobile industry processor interface

MSB most significant bit

MuTRiG muon timing dissolver including gigabit-link

NMOS n-type metal-oxide semiconductor

PCB printed circuit board

PF packet footer

PHY physical layer (bit transfer layer implementation)

PLL phase locked loop

PSI Paul Scherrer Institute

RAW unprocessed image data

RGB colour representation (Red Green Blue)

SCL serial clock line

SDA serial data line

SDR single data rate

SPI serial peripheral interface

UART universal asynchronous receiver transmitter protocol

ULPS ultra low power state

VC virtual channel identifier

VHDL VHSIC hardware description language

WC Word Count

YUV colour representation (Y for luminance U & V for chrominance)

Contents

1 Motivation

In an attempt to find evidence for new physics beyond the Standard Model of particle physics, the Mu3e collaboration is currently prototyping a new pixel detector, featuring bleeding edge technology. The nature of the experiment necessitates tight restrictions on the material budget around the active area, while simultaneously requiring a high rate of extremely precise measurements. The team is pushing the boundaries of HV-MAPS detector technology with every generation of their MuPix[2] sensor, striving for higher resolution and faster readout within the same 50μm formfactor. This development in itself is an impressive feat, but conclusive data demands more accuracy. Employing auxiliary systems, exploiting complementary data and improving the detection precision in software, is squeezing the gap between the observable and its underlying reality.[6] Relating to only a fraction of such a sub-system, this work – while not providing any advances in the semiconductor, much less the physics domain – might seem insignificant, but, in fact, explores the associated technologies and shall provide an informational basis for (and serve as a precursor of-) the implementation of said sub-system.

1.1 The Mu3e Experiment

The Mu3e experiment is a physics experiment located at the Paul Scherrer Institute in Switzerland. It is a collaborative effort of several research groups spanning different institutes in Switzerland, England and Germany - namely the aforementioned Paul Scherrer Institute, the University of Zurich, the University of Geneva, the ETH Zurich, the University of Heidelberg, the Karlsruhe Institute of Technology, The University of Liverpool, the University of Oxford, the University of Bristol, the University College London and the University of Mainz. Mu3e's stated goal is muon decay observation at a high rate (up to $2 * 10^9$ muon decays per second), in search of a specific event that violates the Standard Model of particle physics.[5]

1.1.1 Theoretical Background

Concretely, Mu3e is searching for the so called lepton flavour violating decay

$$\mu^+ \rightarrow e^+ e^- e^+$$

In case of this observation, the standard model of particle physics would be violated and would need to be mended. The characteristics setting Mu3e apart from previous experiments searching for the same decay, is its rate of observable decays. The incidence of a specific decay versus any other decay of the same particle is given by the branching ratio

$$\mathcal{B}(\mu^+ \rightarrow e^+ e^- e^+) < 10^{-12}$$

The Standard Model including massive neutrinos, that accounts for this decay, predicts this ratio at around 10^{-54} while the SINDRUM experiment from 1988 [3] reports no such event in 10^{-12} measured decays. The goal of Mu3e is to restrain the upper bound of the branching ratio to 10^{-16}, thereby eliminating beyond Standard Model theories, that predict a larger branching ratio. In case $\mu^+ \rightarrow e^+e^-e^+$ was detected, the experiment would provide hard data refuting the standard model, instigating further research into the phenomena causing this kind of decay. Mu3e would also give insight into the resulting Leptons' properties, informing new models and experiments.[6]

1.1.2 Experimental Setup

The setup is a tiered system of a muon accelerator, a purpose-built sensor assembly operating in a strong magnetic field inside a controlled atmosphere, high speed custom readout electronics and a compute cluster for data processing. In its current Phase, Mu3e uses the πE5 beam line at PSI, which can produce up to 10^8 muons per second. The muons are guided through a vacuum to a 3.2m long magnet providing a cylindrical field measuring 1T (see figure 1.1). The sensor cage – supporting detector hardware that is layered around the beam target – sits inside the 1m wide inner bore of the 31-ton Solenoid. Inspired by the SINDRUM experiment, the stopping target is a hollow double cone with 70 μm thick walls made of Mylar. Upon impact on this 19mm wide target, the muons can decay into elementary particles, which in turn travel outwards, while following a spiralling path determined by their momentum.[1][12]

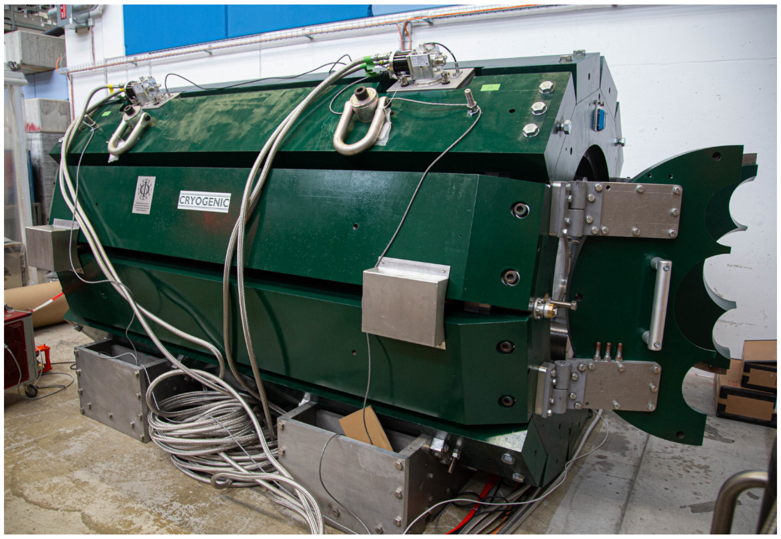


Figure 1.1: Mu3e magnet. (Niklaus Berger 2020)

Measuring the particle paths with extremely high spacial and temporal resolution, is crucial for identifying the particles and the decay, they originated from. Although the particles travel at a high speed, they are prone to deflection caused by any compact solid in their way. To reduce losses in usable data due to this unwanted scattering, the sensor assembly was designed for minimal obstruction of the particle initial scattering vectors. This is achieved through the use of 50μm thin High-Voltage Monolithic Active Pixel Sensors (HV-MAPS) [17] called MuPix, bonded to flexible printed circuits for data readout. These so-called "ladders" are arranged as four nested cylinders around the stopping target, overlapping at the edges to cover any angle perpendicular to the beam. Particles scattering from the target travel through the first two layers of MuPix sensors, a layer of scintillating fibre ("sci-fi") sensors and the last two MuPix layers for a total of four spacial data points as well as a high precision temporal data point from the sci-fi sensor per path. in case the particle trajectory curls back towards the beam, two additional sensor segments were added on both sides of the active part. These "recurl" modules have the same two outer pixel layers, but replace the inner part with scintillating tile detectors. The relative positioning of all detector elements is visualised in figure 1.2. The scintillating detectors are read out using silicon photomultipliers. The entire assembly is contained in a dry helium atmosphere, the MuPix layers being cooled by helium gas flowing through the mounting brackets, over the ladders (see figure 1.3).[1]

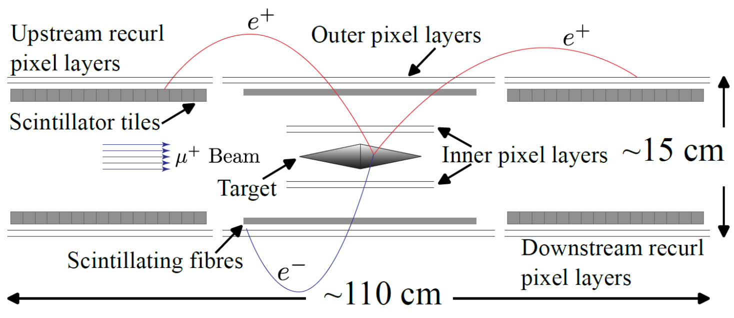


Figure 1.2: Mu3e detector layout. (Data Flow in the Mu3e DAQ, Marius Köppel 2022, Fig. 1)

To accommodate the high rate of muon decay events inside the detector, every part of the system is designed for maximum throughput across physical, electronic and digital domains. At the detector output, the signals cross over into the digital domain, entering the Data Aqusition System (DAQ)[12]: Custom ASICs (MuTRiG for the scintillators, integrated readout electronics on MuPix) retrieve the hit data from the triggerless detectors, labelling and relaying it via electrical low voltage differential signalling (LVDS) link to the front-end boards (FEBs). The front-end boards are FPGA-based (field programmable gate array) to accommodate the high data rate of up to 56.4Gbit per second (for the current stage I of the experiment) and resist the magnetic field. They sort and package hit information, relaying it to the optical switching boards. Through the high bandwidth optical link, the data leaves the magnet, heading for the event filter farm.

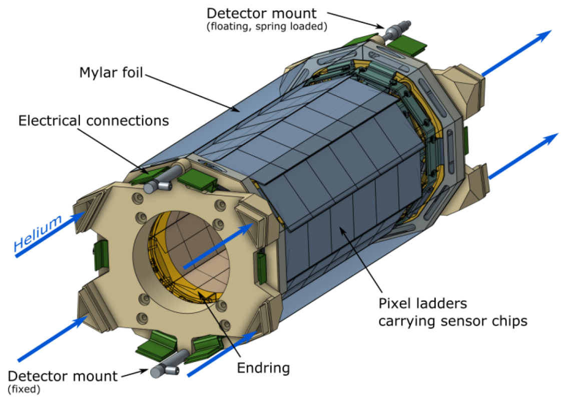


Figure 1.3: Inner pixel layer assembly. (Technical Design of the Phase I Mu3e Experiment, Arndt et. al. 2021, Fig. 7.6)

There, the data is distributed over several GPU-equipped PCs filtering out only the relevant event data. The result is a stream of around 50-100MB per second of selected events, that are stored for later analysis using the MIDAS Software[18](see figure 1.4).

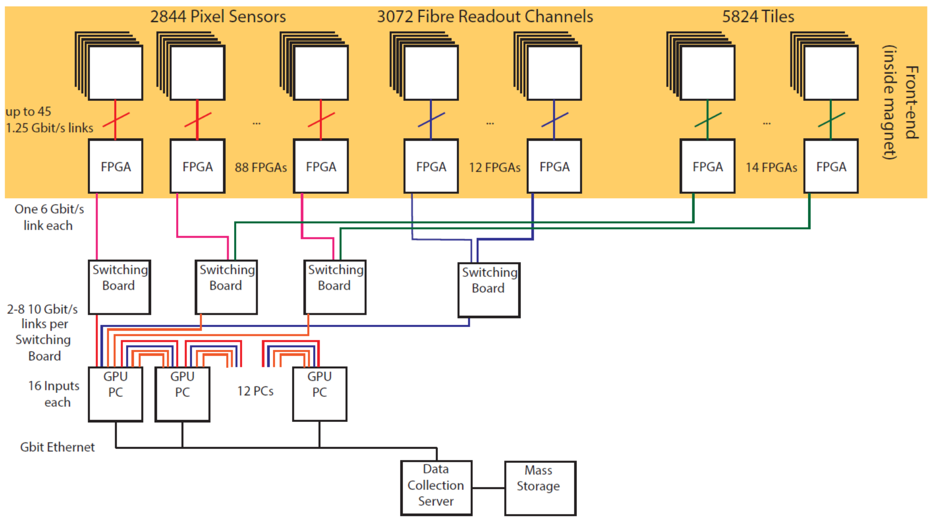


Figure 1.4: Mu3e readout scheme. (Technical Design of the Phase I Mu3e Experiment, Arndt et. al. 2021, Fig. 17.1)

1.2 Camera Alignment System

Measuring the exact timing and position of sensor hits is crucial for reconstructing the particles' exact paths, since they are used for inferring the particles' physical momentum and origin. Due to the material budget requirements of the sensor elements, the MuPix chips are bonded onto a flexible printed circuit. Although the manufacturing of these ladders has been refined multiple times, the exceptionally thin structures can easily deform. To remedy this, as well as thermal- and gravitational influences on the sub-assemblies' relative positions, multiple alignment strategies have been proposed.[19] Track based alignment computes the best fit for a particle's track using all four hits, to calculate each deviation from this "true" track. The MILLEPEDE-II algorithm[4] then minimises the error accumulating all deviations, to obtain global alignment parameters optimising the track fit. This method works well for offsets directed laterally or orthogonally to the sensor plane, but cannot easily account for misalignment of sensor sub-assemblies, called the weak modes. These misalignments consistently allow good track reconstruction, although the sensors are shifted (see Fig 1.5).[8]

As a means to tackling this problem the camera alignment system[19] was proposed.

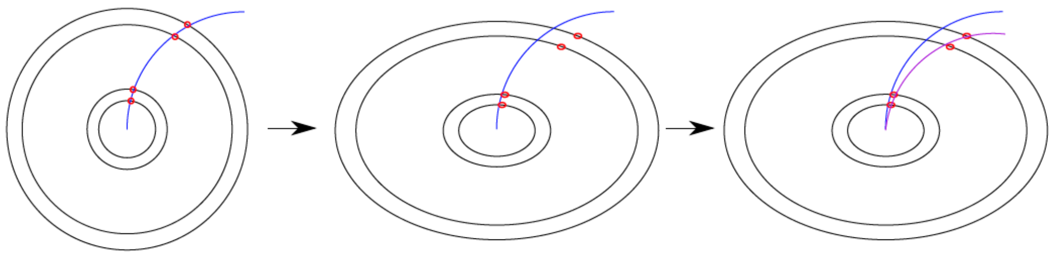


Figure 1.5: Example of weak mode misalignment and track reconstruction at the inner- and outer pixel layers. (Track Based Alignment for the Mu3e Detector, U. Hartenstein 2019, Fig. 5.7).

Instead of relying on the data from the sensors, this method uses LEDs mounted on the sensor cage as well as on the sensor itself to track their relative positioning using cameras. The mounting pattern of the LEDs and cameras allows each camera to capture two other camera positions (marked by two leds mounted on both sides of the sensor sitting on a custom PCB), as well as LED-marked reference points on the sensor assembly as well as the cage (see figure 1.6). The position of these references – relative to the part it is attached to – is known and the distances between them can be extracted from the image. Using this data, the positioning in 3D space of each marked part in the assembly can be calculated using triangulation, allowing greater spacial accuracy in the measurements. Bridging between the cameras and the Mu3e compute cluster will be realised using custom made FPGAs, allowing for great flexibility in terms of mechanical arrangement in the sensor assembly, as well as multiplexing the data lanes – possibly even integrating the control link – for a large reduction in the system's interface fan-out.[7]

2 Technologies

The summaries of the respective technologies detailed below, shall provide a solid basis for future revisions of the proposed CSI-2 bridge implementation. It highlights the

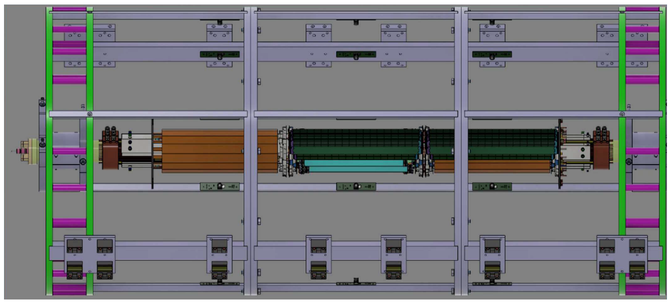


Figure 1.6: Camera positioning in the sensor cage. (Technical Update on the Camera Alignment System, Sophie Gagneur 2023

most relevant aspects of each Protocol, eventually presenting additional information if deemed relevant or for completeness' sake.

2.1 FPGAs

Field programmable gate arrays (FPGAs) are configurable integrated compute units, which can be programmed to function as user defined hardware. They contain generic hardware blocks that can be manipulated with gate-level precision. Combined, these blocks may serve as combinatorial- and/or clocked circuits. These devices are crucial for Mu3e, allowing exceptionally high throughput, as well as cost- and time efficient configuration changes in the data acquisition system (DAQ) while serving as bridge between the MuPix sensor modules and the processing plant, as well as consolidating incoming data for the GPU cluster. Additionally FPGAs come in several "hardened" variations, protecting the circuits from external influences, for instance radiation, electro magnetic interference or extreme temperatures. These properties make them heavily sought-after for use in extreme conditions and high performance systems in general.[12] For the camera alignment system, one or multiple FPGAs assume the role of a bridging component, controlling the cameras' image sensors and relaying their data output to the DAQ, where it is stored for on- or offline analysis. The FPGA for the prototype implementation presented here, is a Lattice Semiconductor MachXO3L DSI breakout board. It was chosen for is close resemblance of the MachXO3LF – which is familiar to the author – and its compatibility with high speed MIPI physical layers (in this case D-PHY). The accessibility of these high speed components turned out to be an unexpected but significant impediment of the project. Lattice advertises the MachXO3L DSI and SMA versions of the same breakout board as having similar capabilities, in that the only difference is which connector is populated. The fact that the two interfaces don't share the same internal capabilities, was not obvious. For this reason the custom adapter card produced for the bridge setup was mended once, supplemented using jumper wires and then revamped, to completely disregard the initial DSI connector. Consulting the Lattice documentation to resolve such ambiguities, is laborious, in that it is distributed over several documents, most of which are neither included in the concerning FPGA's documentation package, nor indexed for the documentation search function on their website. Often times the only way to find a document is searching in online forums or clicking on the reference links in another part of the documentation. The Lattice prototype board will be replaced by a custom Intel MAX10-based board connecting directly to the cameras, potentially adding MIPI bifurcation, as well as the front-end boards.

2.2 I²C

The I²C or Inter-Integrated Circuit standard (also IIC, I2C) was defined in 1982 by Philips Semiconductors (now NXP Semiconductors). It is closely related to the serial peripheral interface (SPI) and universal asynchronous receiver transmitter protocol (UART) and describes a simple solution to allowing data transactions between two or more devices using only two wires and a robust protocol.[10]

2.2.1 Hardware

The two wires are referred to as SCL and SDA, denoting the data and clock lane. Both lanes are bidirectional buses and are required to reliably transmit serial data at 100 kbit/s for Standard-, 400 kbit/s for Fast-, 1 Mbit/s for Fast-mode Plus and 3.4 Mbit/s for High-speed mode. A unidirectional 5 Mbit/s mode called Ultra Fast-mode is also defined in later iterations of the standard. The binary signals may be represented by different voltage levels, depending on the connected devices. This facilitates the usage of NMOS and bipolar logic, opposed to only CMOS which runs on 3.3V direct current. The signal wires are coupled to the supply voltage (V_{DD}) of one of the devices using a pull-up resistor. Since there is no active electronic component regulating the bus voltages, all devices, transmitting to or receiving from the same I²C bus at the same time, must use the same voltage to do so.[10]

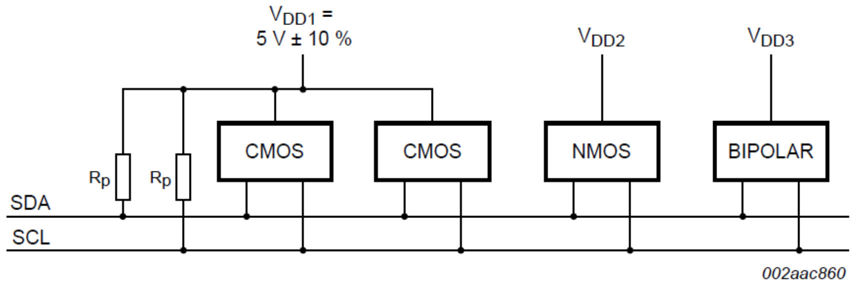


Figure 2.1: Exemplary I²C architecture. (I2C-bus specification and user manual, NXP Semiconductors 2021, Fig.3)

2.2.2 Protocol

The following applies to all modes except Ultra-Fast mode, unless declared otherwise. Any device connected to the bus can send or receive data, but only one can talk at a time. The device initiating a transaction is called the controller and provides the clock- and stop signals for this interaction. All devices have a unique address, which – if received – prompts the respective device (called target) to listen for incoming commands. A targeted device can act as transmitter to the bus, for example when a read command is received. In this case the target sends data synchronized to the controllers' clock signal. The protocol can be classified as packet-switched.[10]
For the purpose of better understanding, a simplified standard operation of the bus using one controller and one target is assumed and subsequently expanded on in the next subsection.
The most essential Signals are the START- and STOP conditions, generated by the controller to initiate or terminate a transaction. When all devices are in idle, the bus lines are both tied to HIGH. To signal a new transaction, the controller pulls SDA low, while holding SCL high for a specified hold time (depending on the configured bus speed, $t_{\mathrm{HD;STA}}$ shall be at least 4, 1.3 or 0.5 µs). Immediately after, the first clock cycle and data bit shall be sent. To end a transaction the controller holds SCL high, while transitioning SDA from LOW to HIGH. The setup time between pulling up SCL and SDA is defined by $t_{\mathrm{SU:STO}}$, and, in the same vein t_{BUF} requires a mandatory buffer time between a STOP condition and the next START. Instead of STOP, a repeated START can be sent by the controller to continue a command that spans multiple transactions or address a different target, without deallocating the bus.[10]
Omitting sub protocols defined within in the payload, data is always transmitted Most Significant Bit (MSB) first. All transactions consist of 8 bit sized bytes followed by one bit that is reserved for acknowledging data reception (ACK is active low). The specification does not limit the amount of 9-bit packages contained in one transaction. Addressing can be realised using the first 7 bits of a transaction to identify a device and the eighth bit (RW bit) to indicate whether the following byte shall be read from or written to this device (LOW indicates WRITE, HIGH indicates READ). In case of a read request by the controller, the roles of transmitter and receiver switch within the same transaction. When a target device receives a transaction with its address and the read flag set, it proceeds to write 8 bits of data – starting from its current device memory pointer – to the data line. The controller therefore releases the data line and continuously provides clock cycles, so the data provided by the target stays in sync. For this reverse data packets, the controller becomes the one to send a LOW Acknowledge bit for any successful transmissions or HIGH for unsuccessful ones (the latter can only be detected by a well defined sub protocol or in case of violation of the I^{2}C protocol).

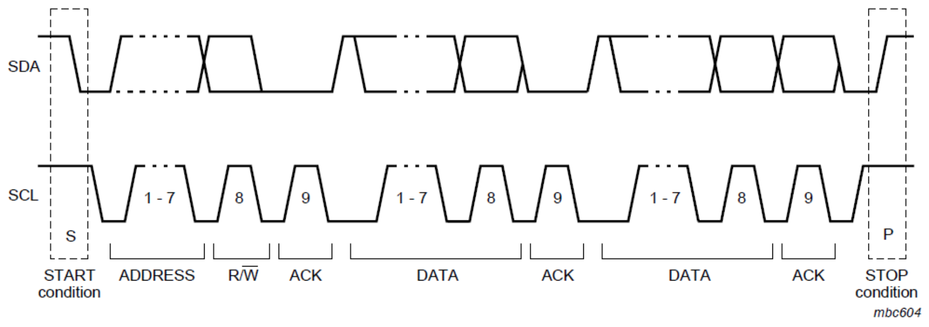


Figure 2.2: I^{2}C transaction timing diagram. (I2C-bus specification and user manual, NXP Semiconductors 2021, Fig.3)

2.2.3 Extended Functionality

The I^{2}C protocol features a variety of optional extensions to facilitate a wide range of configurations.
Since low complexity devices benefit a lot from being able to communicate with a larger system, the protocol can be augmented to fit their abilities. Clock Stretching allows a slower target device to hold the clock line low while processing the bit received on the previous clock HIGH. The controller then has to wait for the target to release SCL, before continuing its transmission. If at least one target device uses clock stretching, all controllers on the same bus are required to support this functionality.
A similar technique is used in multi-controller environments, to synchronise clock signals when multiple transactions have been initiated and compete for the data line. When detecting a falling edge on the clock line, any controller trying to send data must count down its clock LOW time and hold the respective line. Thereby, the controller using the longest LOW time dictates the LOW period of SCL. On the other hand the controller with the shortest HIGH time will be the first to pull the clock down and issue the HIGH period length. In this synchronized state, all active controllers are able to reliably read from and write to the data line. This is mandatory for the arbitration process i.e. negotiating which controller is allowed to write to SDA.
Arbitration is realised as a passive process in each active controller. Adhering to the synchronized clock signal, all controllers try to drive SDA. As soon as one controllers internal data level differs from SDA while the clock is high, it must stop transmitting immediately and – if applicable – switch to target mode. The cancelled transactions shall be repeated after the next STOP signal.
Since the simultaneous activity of multiple controllers can still be prone to logical errors, the START byte and Software Reset extensions are recommended. 10-bit addressing and General Call addresses can also improve system performance, but will not be touched on here.[10]

2.3 MIPI CSI-2

The Camera Serial Interface 2 (CSI-2) is a specification introduced by the Mobile Industry Processor Interface (MIPI) alliance in 2005. It serves to standardise high speed camera interfaces for wide ranging use cases in the embedded space, enabling controlling a camera module and reading out its image data at high speeds. Therefore the protocol is highly flexible as it can be implemented on different physical layers and allows the usage of various pixel data formats to suit the use case perfectly. The physical layer can be implemented as a low power high speed C-PHY or D-PHY link, as well as a more sophisticated A-PHY for longer range transmission or a very low cost MIPI I3C interface. These interconnects are not equally capable in terms of image size or bit depth, so any implementation must maintain backwards compatibility. Additionally the CSI-2 interface includes a separate I^2C bus for image sensor configuration called the Camera Control Interface (CCI). The I^2C and D-PHY interfaces oppose each other in data directionality: While pixel data is transmitted unidirectional from the image sensor to the image processing unit, the camera control signals are issued by the latter. The CCI data flow is bidirectional, since the target device can be read out. In this case the target drives data to the bus, but the rolls of controller and target cannot be reversed in a standard CSI-2 setup.[14]

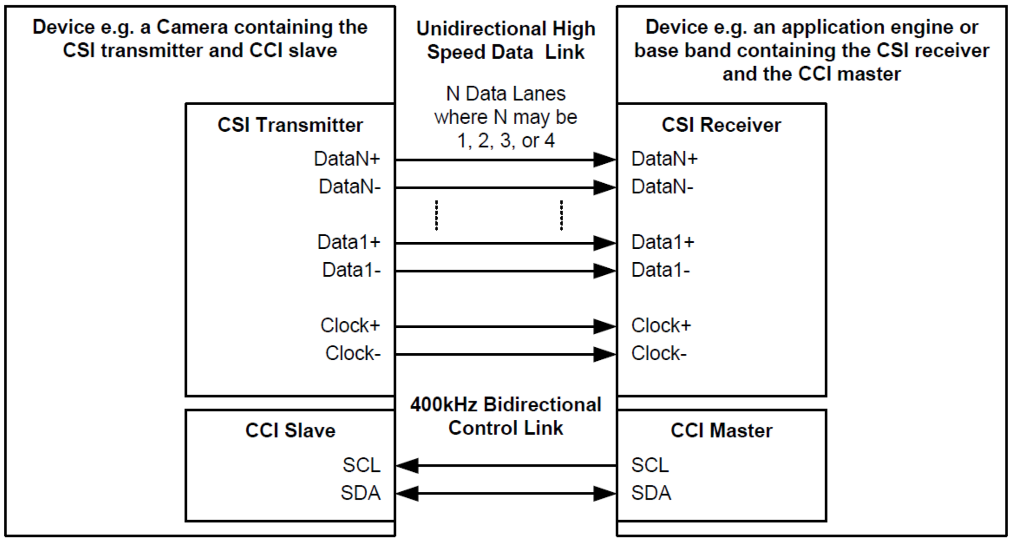

Device e.g. a Camera containing the CSI transmitter and CCI slave
Unidirectional High Speed Data Link
Device e.g. an application engine or base band containing the CSI receiver and the CCI master
N Data Lanes where N may be 1, 2, 3, or 4
CSI Transmitter
DataN+
DataN-
Data1+
Data1-
Clock+
Clock-
CSI Receiver
DataN+
DataN-
Data1+
Data1-
Clock+
Clock-
400kHz Bidirectional Control Link
CCI Slave
SCL
SDA
CCI Master
SCL
SDA

Figure 2.3: CSI-2 Interface. (MIPI Alliance Specification for CSI-2, MIPI Alliance 2009, Fig. 1)

2.3.1 CCI

The camera control interface builds on top of the standard I^2C implementation running at 400kHz and using a single controller, multi target architecture. The target device shall support 7 bit addressing, 8- or 16 bit internal indexing and 8 bit data blocks.

Register memory shall be byte-aligned, to allow sequential operations on multi-byte registers that hold 16-, 32- or 64 bit values. Consecutive read- or write operations can be combined into a single sequential transaction each. In this case the controller starts at the first register address and does not terminate the transaction until all bytes have been transferred. The target device shall increment its internal register pointer accordingly and is required to buffer multi-byte registers as soon as the first byte is addressed. This feature of the target device also allows for implicit addressing in read operations, where the controller issues a read transaction without previously specifying a internal address using a write statement. Otherwise, reading from a random location on the target device starts with a write operation, writing 8 or 16 bits (depending on the register index), thereby setting up the register pointer. This transaction looks like the start of a write operation, but is prematurely interrupted by a repeated Start condition to request an implicit read from the previously specified location.[14][10]

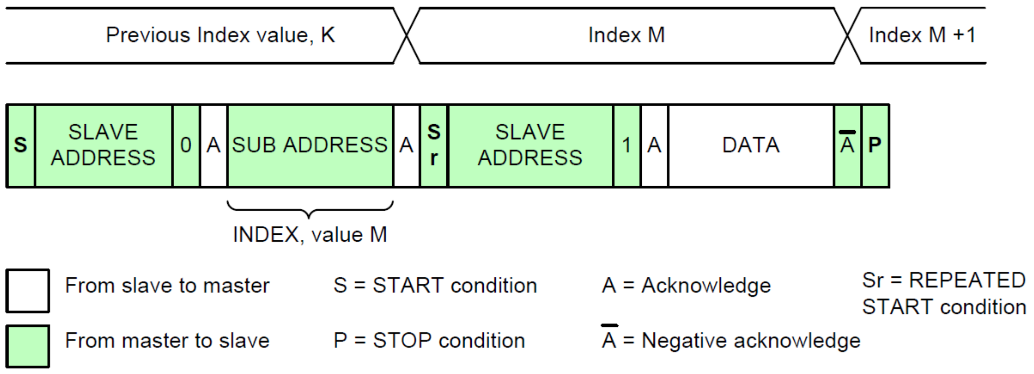


Figure 2.4: Exemplary single read from random location. (MIPI Alliance Specification for CSI-2, MIPI Alliance 2009, Fig. 4)

2.3.2 D-PHY

MIPI D-PHY is a physical layer implementation compliant with CSI-2. It is built on one clock lane and at least one data lanes, each of which use low voltage differential signalling (LVDS) for better electro magnetic interference (EMI) resistance. The two different operating modes defined on the data lanes – called high speed (HS) mode and low power (LP) mode – use different transmission speeds and voltages. The high speed mode is used to transmit large amounts of data in synch with the high frequency clock lane and has a tight voltage swing between its logic levels. It transmits data using double data rate (DDR), where the data lanes are sampled twice per clock cycle. Low power signals are independent of the high speed clock and enable extended functionality (escape mode) via its slower, large swing signals. While the clock lane is always unidirectional, therefore defining the forward direction, both HS and LP signals can be implemented bidirectionally. To manage all possible configurations, a lane module

Prefix	**Lane Interconnect Side**	**High-Speed Capabilities**	**Forward Direction Escape Mode Features Supported**	**Reverse Direction Escape Mode Features Supported**
CIL-	M - Master	F - Forward Only	A - All (including LPDT)	A - All (including LPDT)
	S - Slave	R - Reverse and Forward	E - events - Triggers and ULPS Only	E - events - Triggers and ULPS Only
	X - Don't Care	X - Don't Care	X - Don't Care	N - None
				Y - Any (A, E or A and E)
				X - Don't Care
		C - Clock	N - Not Applicable	N - Not Applicable

Table 2.1: D-PHY lane type descriptors. (MIPI Alliance Specification for D-PHY, MIPI Alliance 2009, Table 1).

hardware component is needed for each end of any signal wire pair, implementing the so called Control and Interface Logic (CIL). Any D-PHY lane configuration can be denoted using standardised descriptors (see table 2.1).[15]

CSI-2 uses CIL-MFEN and CIL-SFEN configurations for TX and RX side of its up to four data lanes as well as CIL-MCNN and CIL-SCNN on the respective sides of its clock lane. This means only unidirectional data transfers are allowed and functionality of the (forward only) escape mode is reduced. In escape mode, the clock lane only supports ultra low power state (ULPS – wherein both signal lines are low and the system is idle until a wake-up signal is transmitted), while the data lanes can also transmit 8 bit trigger codes via the low power signals (The meaning of these trigger codes shall be defined by the protocol layer above D-PHY).
D-PHY lane modules interact using all 4 possible permutations of the two signal states on each line. Since they are interpreted differently between LP and HS mode, 6 distinct signals are defined.

The default mode of the lane module is the control mode, where both signal lines are high. From there the transmitting lane module can request to enter high speed or escape mode using low power signal sequences. The sequence Stop - HS-Rqst - Bridge (LP11 -

State Code	**Line Voltage Levels**		**High-Speed**	**Low-Power**	
	Dp-Line	**Dn-Line**	**Burst Mode**	**Control Mode**	**Escape Mode**
HS-0	HS Low	HS High	Differential-0	N/A	N/A
HS-1	HS High	HS Low	Differential-1	N/A	N/A
LP-00	LP Low	LP Low	N/A	Bridge	Space
LP-01	LP Low	LP High	N/A	HS-Rqst	Mark-0
LP-10	LP High	LP Low	N/A	LP-Rqst	Mark-1
LP-11	LP High	LP High	N/A	stop	N/A

Table 2.2: D-PHY lane state description (MIPI Alliance Specification for D-PHY, MIPI Alliance 2009, Table 2).

LP01 - LP00) signals the receiving lane module to switch into HS receiver mode, before the transmitting module starts sending HS data. High speed data is sent in multiples of 8 bit and surrounded by starting and ending sequences of HS signals. The former is used to synchronise the receiving modules HS receiver, while the latter initiates the switching back to control mode on the RX side. If instead, the control mode is exited using the LP11 - LP10 - LP00 - LP01 - LP00 sequence, the lane modules switch to Escape mode. 8-bit command patterns define Escape mode functionality. The transmitter can issue low power data transmission (LPDT), ultra low power state and send or clear user defined trigger signals. LPDT enables sending 8-bit packets using differential signalling that allows asynchronous pauses using LP00 and is terminated by LP11. If a data lane is bidirectional, the sequence LP11 - LP10 - LP00 - LP10 - LP00 triggers turnaround mode, where transmitter and receiver switch roles, while the clock lane always remains in forward direction.[15]

2.3.3 Multi Lane Operation

CSI-2 multi-lane distribution and merging follows the round robin principle, where N bytes from a transmission are buffered so that each of the N transmitters simultaneously start transmitting one byte each until the entire message is consumed. The end of transaction signals may arrive asynchronously in case the message size can not be divided by N. Lane distribution shall accommodate for disconnected lanes by skipping them, allowing robust data transmission at the cost of throughput.[14]

2.3.4 Pixel Data Packets

The data packets passed to the physical layer follow a low level protocol, using labelled payload data and defect detection measures, to allow reliable transmission and rapid processing of the data. All packet transmissions must be separated by a period of low

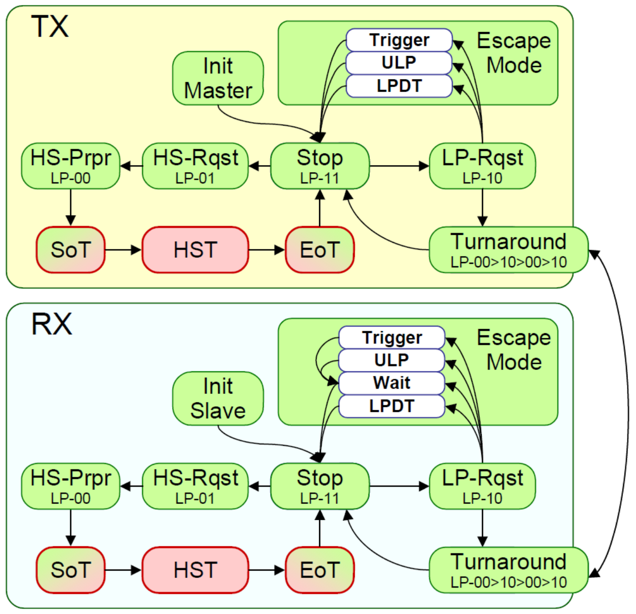


Figure 2.5: D-PHY data lane module operation flow diagram (MIPI Alliance Specification for D-PHY Version 1.00.00, MIPI Alliance 2009, Fig. 24)

power state, which may be timed on the transmitter site, to allow for easier line- or frame synchronisation on the receiver side.

The payload itself consists of multiple 8-bit data words containing YUV-, RGB-, RAW- or user defined data. It is preceded by a packet header containing an 8-bit data identifier (DI), a 16-bit size descriptor (word count WC) and an 8-bit error correction code (ECC), ensuring the headers' integrity. A 16-bit checksum covering the entire payload, called the packet footer (PF), is trailing the data.

Short signalling package formats are also defined, only containing the data identifier, 16 bits of data and an 8-bit ECC. The short packet data field can be used to transmit user defined data, as well as frame- or line- number for synchronisation packages. The processing of any packet is guided by its data identifier, which consists of two bits for the virtual channel identifier (VC) and 6 bits for the data type (DT). While the data type only describes how to interpret the packages' data field, the virtual channel identifier can be used to send multiple data streams via the same PHY.[14][15]

3 Implementation

This chapter describes a prototype VHDL implementation of a CSI-2 bridge, submitted with this document. The aim was to apply the knowledge gained from the previously discussed protocol documentations and examine their real world pertinence. Due to

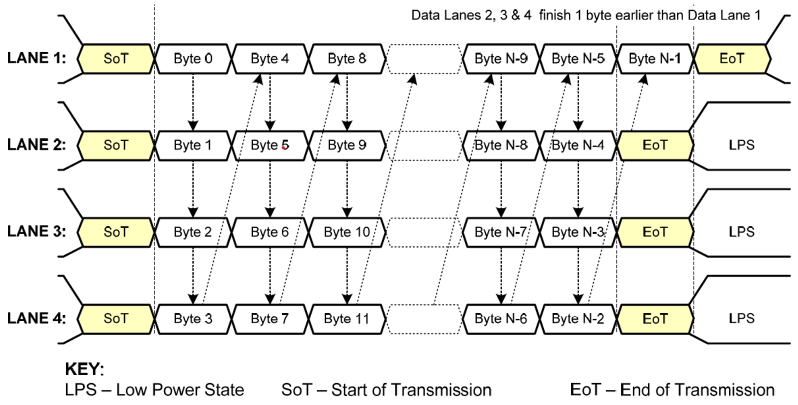


Figure 2.6: Lane distribution across four data lanes. (MIPI Alliance Specification for CSI-2, MIPI Alliance 2009, Fig. 24)

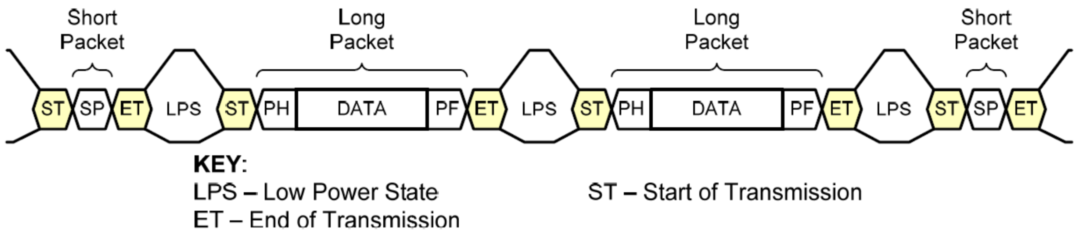


Figure 2.7: CSI-2 packet spacing. (MIPI Alliance Specification for CSI-2, MIPI Alliance 2009, Fig. 29)

time constraints, the development followed a rapid revision cycle, replacing entire components on the fly instead of developing a rigorous specification for the bridge system software first. The process followed the control dependencies, from issuing the initialisation sequence to receiving the first image data, testing each component before building on top of it. This strategy entailed favouring full integration of the image data lanes and camera start-up sequence, over developing an expansive feature set and a convenient front end for the sensor control interface, all of which would be relevant for the final camera alignment system.[19]

During of the implementation, two major reworks have been conducted. Initially an I²C component [9] from the Lattice IP catalogue was used, which provided a register based interface for connecting a micro controller (MCU). Following this approach, the "CCI encoder" entity was envisioned to relay transactions between the front-end boards (FEBs) and the Lattice component, which in term controlled the image sensor. While studying the IMX219 initialisation sequence from the specification, the conclusion was made that continuous transactions might be needed to send two bytes of data, addressing only the upper byte in 32 bit camera registers.[11] Since the Lattice component obfuscated the inner workings of the I²C controller, requiring another logical translation layer to invoke the correct signalling modes, it was promptly replaced by a more user-friendly implementation. This component was implemented by Scott Larson and

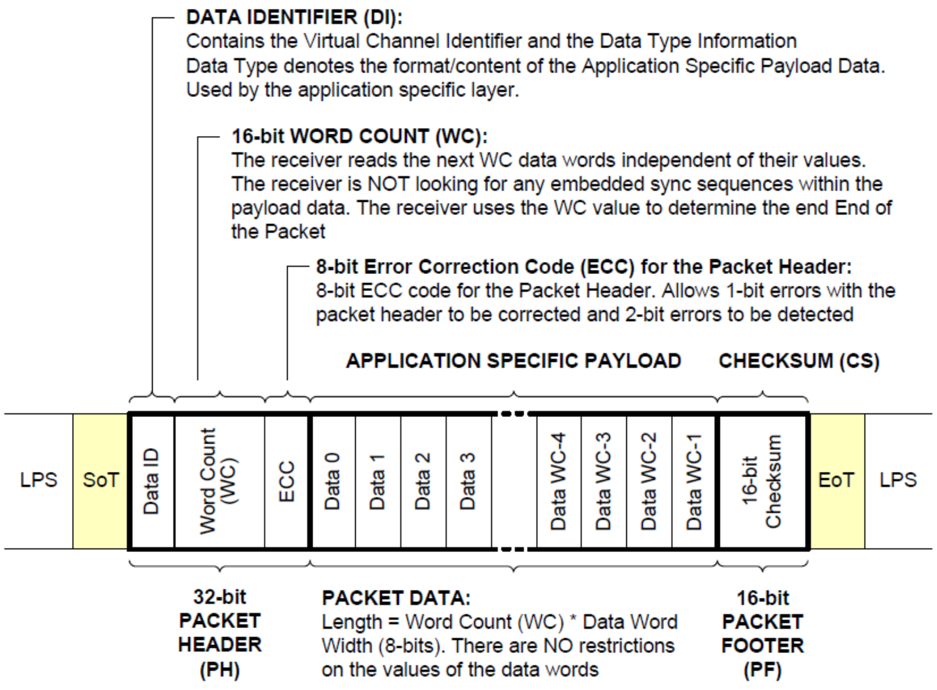

DATA IDENTIFIER (DI):
Contains the Virtual Channel Identifier and the Data Type Information
Data Type denotes the format/content of the Application Specific Payload Data.
Used by the application specific layer.
16-bit WORD COUNT (WC):
The receiver reads the next WC data words independent of their values.
The receiver is NOT looking for any embedded sync sequences within the payload data. The receiver uses the WC value to determine the end End of the Packet
8-bit Error Correction Code (ECC) for the Packet Header:
8-bit ECC code for the Packet Header. Allows 1-bit errors with the packet header to be corrected and 2-bit errors to be detected
APPLICATION SPECIFIC PAYLOAD
CHECKSUM (CS)
LPS
SoT
Data ID
Word Count (WC)
ECC
Data 0
Data 1
Data 2
Data 3
Data WC-4
Data WC-3
Data WC-2
Data WC-1
16-bit Checksum
EoT
LPS
32-bit PACKET HEADER (PH)
PACKET DATA:
Length = Word Count (WC) * Data Word Width (8-bits). There are NO restrictions on the values of the data words
16-bit PACKET FOOTER (PF)

Figure 2.8: Long packet structure. (MIPI Alliance Specification for CSI-2, MIPI Alliance 2009, Fig. 30)

published on the digikey forum in 2021. Since it had already been used in other projects in the mu3e system, the code was reviewed and deemed fit. This meant that the CCI component underwent a significant rewrite, adjusting to the new interface, now being able to manipulate the I^{2}C controllers behaviour more directly (although a multi-byte transaction was never implemented, since single writes to the lower byte register addresses worked as well).[13][10]
The second component replacement was not a rewrite, rather replacing already written code with a self-contained Lattice component[16]. The image data pipeline consisting of a D-PHY lane management component, a D-PHY HS receiver and a clock synchronisation module for a lane aligner have been deprecated due to time constraints. All of these components lacked proper testing, some even missing features, therefore finishing their integration in time was deemed unrealistic. The decision was confirmed, when better understanding of the FPGA's capabilities lead to the realisation, that at least the HS receivers DDR interface must have been a Lattice IP component either way. This stems from the fact, that DDR data must be sampled on the leading-, as well as the trailing edge of the clock signal, while the expression

```
if (rising_edge(MIPI_Clock_in) or falling_edge(MIPI_Clock_in)) then
```

could not be synthesised. Another option apart from using Lattice's implementation relying on the FPGA's DDR primitives, was to not use the dedicated MIPI ports and routing both LVDS signals from the differential clock pair to the DDR component, but this would take even more time and probably break due to timing or singnal integrity issues, additionally defying the purpose of using a MIPI D-PHY capable FPGA. The Lattice component replacing the image pipeline contains a D-PHY HS receiver, data alignment stages and a packet decoder, that transforms the incoming high speed serial data and presents it at its slower clocked parallel interface. The error correction implementation was salvaged and attached to the Lattice component. The resulting prototype implementation of the entire CSI-2 bridge is detailed below.[9][16]

3.1 Architecture Overview

The proposed MIPI CSI-2 to parallel bridge is composed of 4 main components and can be subdivided in two distinct data pipelines. Firstly the image data pipeline, which consists of a PHY receiver and CSI-2 packet decoder, followed by an error correction module. And secondly the control instruction pipeline, integrating the standard I^{2}C controller with the additional features of the CCI broker, allowing communication between the sensor module and a host system. In the image data pipeline, the data flow is straight forward, from the sensor to the host. Its pupose is modifying the incoming high speed PHY transactions into a slower clocked parallel image format. The ECC component is used to check for transmission errors within the PHY transaction data. The control instruction pipeline is not only buffering transaction data in both directions using the FIFO queues, but also provides a library of human readable commands, that can then be processed and sent to the sensor module via I^{2}C. This also allows reading out sensor data, if desired. The data flow directions as well as the main components relations to each other are visualised in figure 3.1.[16]

3.2 CCI

At the heart of the proposed MIPI CSI-2 bridge lies a I^{2}C control unit called the camera control interface (CCI). It is responsible for initializing and relaying commands to the camera module via a dedicated bus on the CSI-2 interface and providing a higher level interface to issue said commands.

3.2.1 I^{2}C Controller

The first step in developing a working CCI controller was to obtain an I^{2}C controller that can send commands to and read out data from the camera module, following its internal protocol. Since I^{2}C is a widely used Protocol, especially in embedded environments, VHDL source code for this component is readily available.
Originally an I^{2}C controller from the Lattice intellectual property (IP) catalogue was considered to work in tandem with a broker, that would format the requests accord-

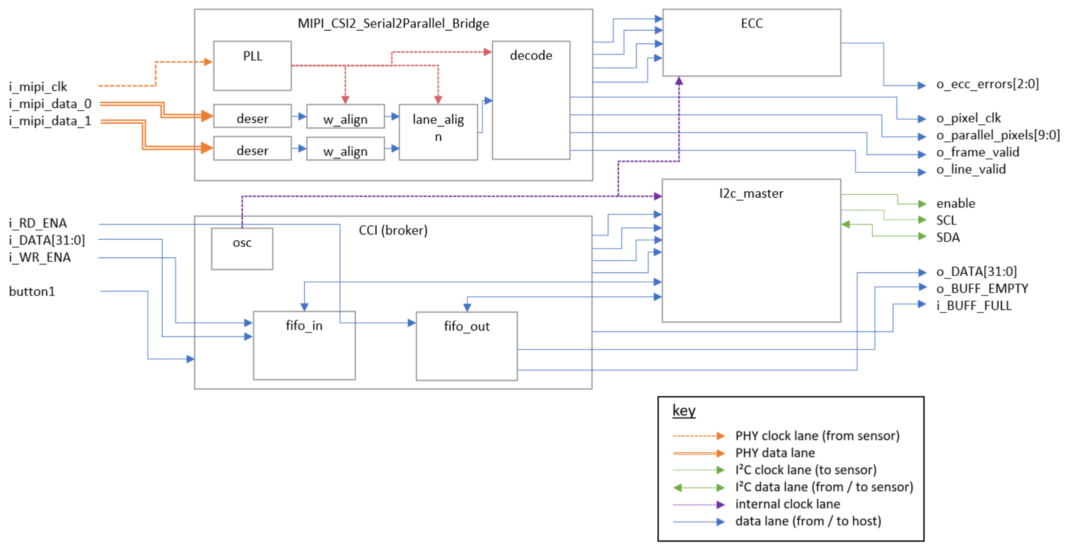


Figure 3.1: Block diagram of the proposed CSI-2 bridge.

ing to the target specification. The Lattice IP component is feature rich, and offers a register based interface, that allows it to be controlled from a different clock domain. Unfortunately the code is packaged in several different sub components, which leads to a large amount of boiler plate code and makes tracing the control flow a convoluted process.[9]

Another fitting VHDL implementation for an I^{2}C controller has been published by Scott Larson and undergone multiple revisions.[13] The feature set of this controller has been reduced to avoid any overhead not contributing to the core functionality of the bus system. Therefore any features serving multi controller bus systems (clock synchronisation, arbitration, START byte and software reset) were deprecated in favour of reduced size on chip and better readability.

The latter option has been elected to be the starting point of this implementation. Compared to the former, its reduced complexity is a benefit for the project, since it makes the code much more readable – and arguably more maintainable – without diminishing functionality in any way. This controller has been configured to use the 400kHz Fast mode.

The I^{2}C controller consists of two finite state machines (FSMs) and a piece of combinatorial logic: One FSM generates an internal- (`data_clk`) and a quarter of a cycle delayed bus clock (`scl_clk`) from the user specified frequencies. It works together with the combinatorial part to generate START or STOP signals and drive the clock and data lanes according to the I^{2}C protocol. The other FSM manages the interaction with the user code using busy and error signals and feedig the transaction data from its inputs to the signal generator mentioned above.[9]

As long as the enable port (`ena`) is low, the management controller is in its *ready* state and polls said register using the user supplied system clock. When `ena` changes to high, the `busy` flag is asserted, signalling the controllers activity to the outside while latching 8 bits for the target address and the read/write flag (`addr`, `rw`), as well as 8 bits of data to be sent (`data_wr`). After the first bit (MSB) of the address is loaded into the internal SDA register `sda_int` and the bus clock is enabled in the *start* state, the *command* state loops, to feed the remaining address bits to the signal generator. On the ninth bit, the controller expects the target to pull the data line low to acknowledge successful reception of its address and RW bit. Therefore the SDA line is released and the FSM switches to *slv_ack*1, probing SDA on the trailing edge of the internal clock (i.e. at the peak of the bus clock) and signals `ack_error` if necessary. On the next leading edge, SDA is freed or prepared with the fist bit of the data byte, depending on the RW bit. The read state *rd*, again loops for 8 bits, latching the SDA values to `data_rx` (sampling at the peak of `scl_clock`). On the ninth bit, the controller sends a NACK signal to end the transaction, unless the user code holds `ena`, `addr` and `rw` as is, to issue another read from the same address. In any case, after a read command, the FSM switches to *mstr_ack*. The write state *wr* on the other hand, just feeds each `data_tx` bit to SDA and continues to *slv_ack*2. *mstr_ack* and *slv_ack*2 effectively do the same thing: if `ena` is HIGH, they release the `busy` flag to signal the user code readiness for receiving new commands, compare the new command with the old one and either continue the ongoing continuous one or initiate a new transaction via the *start* state. If `ena` is LOW, both go to the *stop* state, that generates the respective signal and resets the machine to *ready*.[9]

3.2.2 CCI Broker

The CCI broker is the top level component of the bridge system. It is fairly independent from the image data processing unit (D-PHY receiver and ECC), passing through any MIPI data unaltered. The main function of the broker is to operate the I^{2}C controller, communicate with the outside world using a FIFO stack as well as automated initialization of the camera module and its continuous operation. The host interacts with the FIFO interface to issue CCI commands as per the sensors specification. Eventual read-data is then returned to a separate queue ready to be retrieved by the host. The FIFO component was generated from the Lattice IP core catalogue and is treated as a black box.[9]
The central part of this component is another state machine, operating on the local variable `instruction_ID`. It reaches from 0 to 4 and only ticks up, when the I^{2}C controller sends a `busy` signal. This ensures, that any changes to the signals on the controller are made while it (the controller) is occupied and therefore does not sample those inputs. Every write transaction is 32 bits long, read transactions are 40 bits long. the initial 7 bits contain the camera modules' address, followed by a read/write bit, a 16 bit internal register address of the camera and 8 bits of write- or 16 bits of read data.

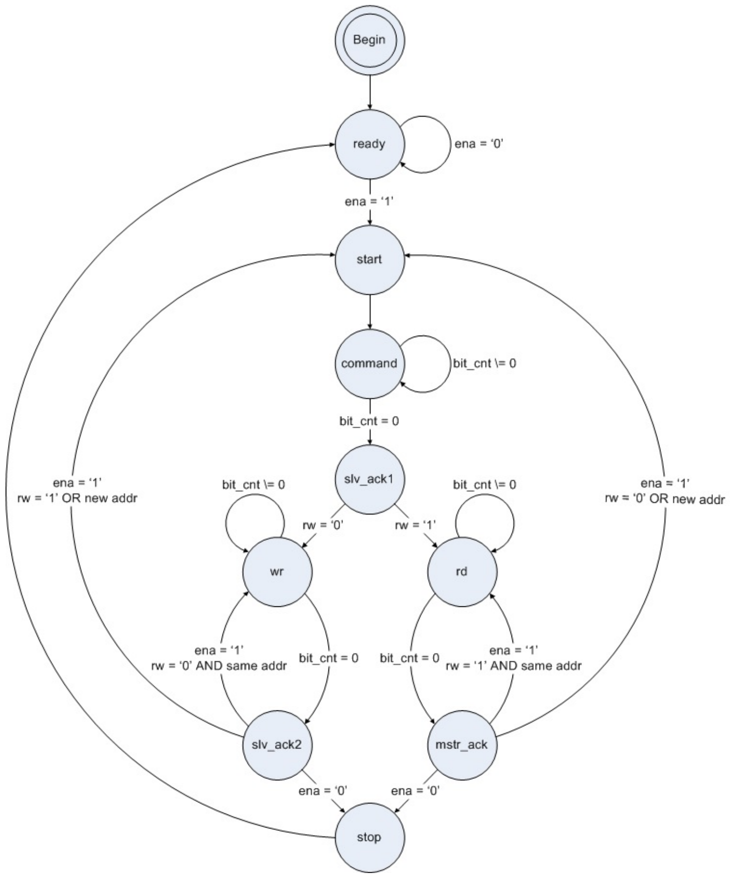


Figure 3.2: I^{2}C controller state machine. (I^{2}C controller documentation by Scott Larson 2021, Fig. 2, `https://forum.digikey.com/t/i2c-master-vhdl/12797`)

`instruction_ID = 0` is the idle state, waiting for a timed internal pulse or an external start signal. In state $IID = 1$ the address of the target, the RW bit and the first half of the register address are defined and the `i2c_ena` flag is set to actuate the I^{2}C controller. The target address is hard coded to the cameras standard address (see section 3.2.3) and the RW bit is always zero, to force direct addressing of the registers. From now on, the broker polls the controllers' `i2c_busy` flag for `instruction_ID` enumeration. As soon as the first half of the register address has been sent, the broker advances to state $IID = 2$ and the second half is written to `i2c_data_wr`. On the next `i2c_busy` signal, the RW bit is overwritten and – depending on the operation – 8 bits of data are written or read using the respective registers. In the last state $IID = 4$, the last 8 bits are read from the bus and sent to the output FIFO, in case of a read operation. Additionally, internal counters and flags may be manipulated here. The I^{2}C controller is disabled using the `i2c_ena` flag and the state machine is reset to $IID = 0$.
The broker's core functionality has been expended upon using internal counters and trigger signals to enable timed periods between instructions, and switching between the FIFO queue and internal arrays for standardized instruction bursts.

3.2.3 CCI Constants

A constants library was added to provide a list of available register addresses on the IMX219 sensor module[11]. This list is not exhaustive, but enables initialisation and basic configuration settings. Two Arrays of I^{2}C commands are defined here: One for initialising the sensor module and one for prompting the sensor to send the next frame. The latter sequence could be used to set basic sensor settings dynamically, for example to adjusting for changes in lighting. The sensor module's target address is hard-coded here, as well as read- and write prefixes. Combined with the numerous register addresses and a hard-coded 8 bit value for each write operation, the transaction arrays are assembled. The last transaction of the initialisation sequence writes to the `MODE_SELECT` register, initiating image data streaming.[11]
Multiple settings of the sensor influence each other significantly, i.e. choosing a larger image size requires a reduction of frames per second or an increase of the internal clock frequency (if not already at the limit INCK = 27MHz). Changing The main clock requires the user to also adjust the pixel- or data rates of the system, using the internal clock dividers and PLL settings or changing the binning or sub-sampling settings. Reconfiguring the D-PHY link or the pixel data format can also be an option to accommodate a higher data rate. The relations between the different clock domains and data rates are outlined in figure 3.3.
To better understand the initialization sequence of the camera module, listening in on an actual start-up conversation between a Raspberry Pi model 3B and the Raspberry Pi Camera module V2 was deemed a fitting approach. The data used, was captured using an inexpensive USB logic analyser in combination with the Saleae logic suite, version 1.2.40. The software is meant for high level digital signal analysis and works well with the relatively slow I^{2}C protocol implementation used with the raspberry pi camera module. The transaction data was saved in CSV format, allowing easy editing. It was verified using a Rohde & Schwarz RTO64 oscilloscope and cross-checked against an existing IMX219-compatible library. The oscilloscope data was not used, since figuring out which acquisition settings could capture more than 70 transactions in one measurement was unjustifiably involved considering the data was the same. A cleaned up version of the original Saleae logic data can be found in appendix A.[11]

3.2.4 Initialization Sequence

On power-up the broker component will automatically generate an internal reset signal to initialise itself. This is necessary to start an automated set up sequence that pushes the hard coded initialisation array to the FIFO stack. As soon as the FIFO is filled, the broker switches to normal operation and detects the data waiting to be transmitted. The stack is then processed one by one, adhering to the piping process described in section 3.2.2, until the `in_buff_empty` signal from the `fifo_in` component switches to

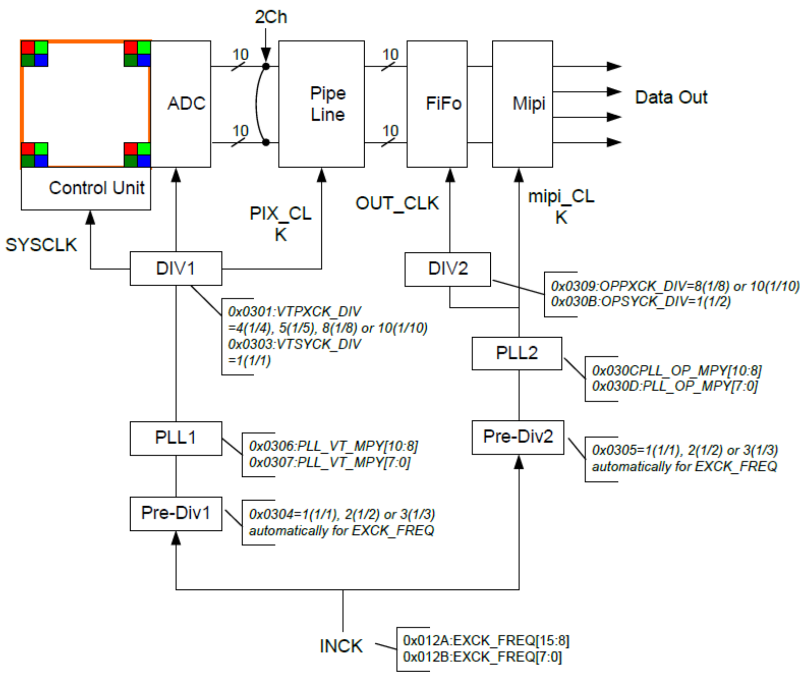


Figure 3.3: Clock system block diagram (Sony IMX219PQH5-C data sheet, Fig. 43)

HIGH. The flag `setup_complete` is set after sending the last transaction, allowing the broker to switch into streaming mode, activating the frame prompt loop.

3.2.5 Frame Prompt

While the sensor module is streaming image data to the D-PHY receiver, the CCI is responsible for switching basic image settings – i.e. analogue gain – on the fly. The respective sequence of transactions is stored in the `REPEAT_PER_FRAME_ARRAY` and sent each frame using an internal timer.[11]

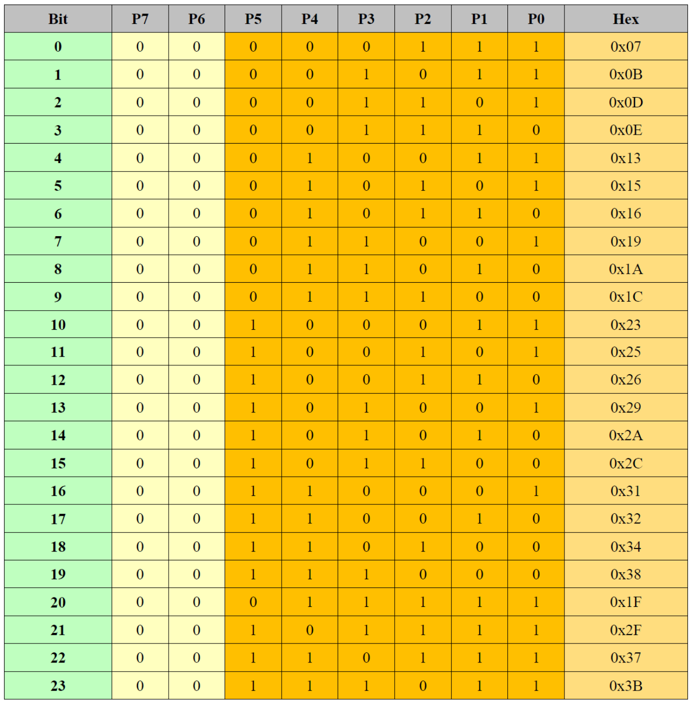
3.3 D-PHY Reciever

The D-PHY receiver is integrated with a CSI-2 lane management module and a user friendly front end, that splits the incoming CSI-2 serial data into a parallel stream. The entire component (called `MIPI_CSI2_Serial2Parallel_Bridge`) has been obtained from the Lattice IP catalogue. Its top level component is only available in Verilog, while the sub-modules have been regenerated as VHDL source code using the Lattice IP-Express tool[16]. The bridge component was modified to accommodate for two PHY lanes, decode the incoming packages as RAW10, and provide an interface for error detection in the header. The PHY implementation uses a deserialisation- and two alignment stages to buffer the incoming packets into word- and lane aligned 8 bit streams clocked at one fourth of the incoming data rate. If the sensor module drives the PHY using a free running clock, the PLL (phase locked loop) sub-component generates the slower pixel clock, that drives the tiered buffer stages. Otherwise the user logic must provide a continuous clock signal that matches the PHY speed, to keep the pixel clock running in case the PHY switches to low power mode. The first stage is a double data rate (DDR) receiver primitive, that converts 8-bit chunks from the fast clocked data lane and distributes them as single data rate (SDR) data over 8 parallel lines. The next two stages synchronise the 8 bit parallel data, aligning it to one 8-bit word per each lane and then synchronising the lanes to each other, using the slower pixel clock originating from the PLL. On detection of a PHY sync sequence on the first lane, the first 32 bit block of every lane is captured to determine the data type and length of the respective data packets. If the format matches the component's internal setting (specified on generation), the pixel data is unpacked and presented at the parallel output pins. Using the pixel clock in combination with the frame valid (`fv`) and line valid (`lv`) ports, the user logic can read the image data from the exposed interface (10-bit parallel for RAW10).[16][14]

3.4 Error Detection

The error correction component implements the error correction code (ECC) generation specified in [14] to detect and – if possible – recover bit flips in the packet header. This code is also calculated on the transmitting side of the PHY and sent as the last eight bits of the header for verification. To generate an ECC the leading 24 bits of the packet header are mapped linearly to a space of 5 bits words, encoding them such that each data bit is covered by any code bit having the same index as the ones of the data bit's index' binary representation. This means, that any single bit flip in the 24 data bits generates a unique pattern diverging from the mapping. This kind of ECC is called Hamming code, honouring Richard W. Hamming, who developed the encoding strategy in 1950. To avoid two bit flips causing the same pattern as a different single bit flip, an additional parity bit is generated for the code word. The resulting 6-bit ECC can be used to detect single bit errors and correct them reliably, as well as detect multiple bit errors.[14]

The implementation uses the transformation matrices given in the CSI-2 documentation to calculate the ECC. The difference between the original ECC and the one calculated by the receiver is called syndrome and encodes the flipped bits' positions. If the syndrome can be matched to a 1-bit error syndrome, the corresponding bit can be flipped back and the header error is mended. Otherwise multiple bit errors occurred and a indicator flag is set.



Bit	P7	P6	P5	P4	P3	P2	P1	P0	Hex
0	0	0	0	0	0	1	1	1	0x07
1	0	0	0	0	1	0	1	1	0x0B
2	0	0	0	0	1	1	0	1	0x0D
3	0	0	0	0	1	1	1	0	0x0E
4	0	0	0	1	0	0	1	1	0x13
5	0	0	0	1	0	1	0	1	0x15
6	0	0	0	1	0	1	1	0	0x16
7	0	0	0	1	1	0	0	1	0x19
8	0	0	0	1	1	0	1	0	0x1A
9	0	0	0	1	1	1	0	0	0x1C
10	0	0	1	0	0	0	1	1	0x23
11	0	0	1	0	0	1	0	1	0x25
12	0	0	1	0	0	1	1	0	0x26
13	0	0	1	0	1	0	0	1	0x29
14	0	0	1	0	1	0	1	0	0x2A
15	0	0	1	0	1	1	0	0	0x2C
16	0	0	1	1	0	0	0	1	0x31
17	0	0	1	1	0	0	1	0	0x32
18	0	0	1	1	0	1	0	0	0x34
19	0	0	1	1	1	0	0	0	0x38
20	0	0	0	1	1	1	1	1	0x1F
21	0	0	1	0	1	1	1	1	0x2F
22	0	0	1	1	0	1	1	1	0x37
23	0	0	1	1	1	0	1	1	0x3B

Figure 3.4: Hamming code generation matrix (MIPI CSI-2 specification Version 1.01.00 r0.04 2-Apr-2009, Table 5)

4 Conclusion and Outlook

The resources aggregated here provide a thorough introduction into the different protocols at play in the camera alignment system. Any part essential for the core functionality is examined, in some cases expanded upon, supporting refinements and extensions of the bridge system's features. This documentation in combination with the proposed bridge implementation, serves as a strong foundation for future development of the camera alignment system.

The implementation – although not perfect – provides a minimal working example of both main signal paths between the camera sensor and the outside world. Repeated testing cycles using the oscilloscope, showed promising results concerning the core functionality of the system, even though no rigorous edge case testing has been conducted. One such edge case, that is currently unaccounted for, would be the FIFO queues filling up. Here, a smarter internal control system, combined with an improved error signalling to the host, would be advisable. Maybe the errors could be integrated in the output FIFO data stream, encumbering the decoding handling on the host, as a trade-off for fewer directly connected data lanes (which might be more relevant, considering the spacial restrictions in the sensor assembly).[1][12] Extending the CCI constant library and extending the CCI state machine to allow multi-byte transactions are obvious improvements, allowing more access to the image sensor's inner mechanisms, although they are technically not necessary for standard operation (writing basic parameters like reso-

lution works already). The data flow between the input FIFO and the I^{2}C controller should allow tighter timing, but for the moment, the safer implementation was preferred over a faster, more fragile one.
Since tight timing is of great concern for the larger system, and the CSI-2 standard is widely adopted, looking into the capabilities of other sensors allowing for faster frame rates using the same resolution might be interesting, in case online usage of the alignment system is desired.[11] For offline alignment, a higher resolution would make measurements more precise. Both should only require the adaptation of the constants library to represent the different register addresses. For more accurate positioning of the sensor module, integrating the sensors carrier PCB into the LED PCB might result in better system performance, while also opening up the option of using all 4 PHY lanes instead of only 2, broken out by the Raspberry Pi module.[11] Combining CSI-2 image data streams through an additional arbitrator board exploiting the virtual channel functionality might also reduce cable clutter. Unfortunately the Lattice image pipeline does not support virtual channel functionality, although it discriminates data streams for their data type, allowing quasi virtual channels by mixing one data type per sensor on the same PHY.[14]
On the other end, the interface on the front end boards, as well as a means of transmitting the image data to the compute cluster remain to be implemented. The development of these two protocols alongside each other might be beneficial to their overall flexibility.[1]
Ultimately the presented implementation servers as a robust starting point for future readout software, representing the precursor of much more that is to come.

References

[1] K. Arndt et al. "Technical design of the phase I Mu3e experiment". In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 1014 (2021), p. 165679. ISSN: 0168-9002. DOI: `https://doi.org/10.1016/j.nima.2021.165679`. URL: `https://www.sciencedirect.com/science/article/pii/S0168900221006641`.

[2] Heiko Augustin et al. *MuPix10: First Results from the Final Design.* 2020. DOI: `10.48550/ARXIV.2012.05868`. URL: `https://arxiv.org/abs/2012.05868`.

[3] W. Bertl et al. "Search for the decay $\mu^+ \to e^+e^+e^-$". In: *Nucl. P* B 260.1 (1985), pp. 1–31. ISSN: 0550-3213. DOI: `DOI:10.1016/0550-3213(85)90308-6`. URL: `http://www.sciencedirect.com/science/article/B6TVC-4719KXC-164/2/f334ec2ed39c5c4c06828b388e60f18a`.

[4] V. Blobel. "Software alignment for tracking detectors". In: *Nucl. Instrum. Meth.* A566 (2006), pp. 5–13. DOI: `10.1016/j.nima.2006.05.157`.

[5] A. Blondel et al. "Research Proposal for an Experiment to Search for the Decay $\mu \to eee$". In: *ArXiv e-prints* (Jan. 2013). arXiv: `1301.6113 [physics.ins-det]`.

[6] A. Blondel et al. "Letter of Intent for an Experiment to Search for the Decay $\mu \to eee$". In: (2012). Available from `https://www.psi.ch/mu3e/documents`.

[7] Sophie Gagneur. *Technical Update on the Camera Alignment System.* 2023.

[8] U. Hartenstein. "Track Based Alignment for the Mu3e Pixel Detector". PhD thesis. Mainz University, 2019.

[9] *I2C Master Controller.* RD1005. Lattice Semiconductor. 2015.

[10] *I2C-bus specification and user manual.* um10204. Rev. 7.0. NXP Semiconductors. 2021.

[11] *IMX219PQH5-C.* E13Y12F44. Sony.

[12] Marius Köppel. *Data Flow in the Mu3e DAQ.* 2022. DOI: `10.48550/ARXIV.2208.13508`. URL: `https://arxiv.org/abs/2208.13508`.

[13] Scott Larson. $I2C_masterdocumentation$. Accessed March 3, 2023. `https://forum.digikey.com/t/i2c-master-vhdl/12797`. 2021.

[14] *MIPI Alliance Specification for Camera Serial Interface 2 (CSI-2).* Version 1.01.00 r0.04. MIPI Alliance. 2009.

[15] *MIPI Alliance Specification for D-PHY.* Version 1.00.00. MIPI Alliance. 2009.

[16] *MIPI CSI2-to-CMOS Parallel Sensor Bridge.* RD1146. Version 1.00.00. Lattice Semiconductor. 2016.

[17] I. Perić. "A novel monolithic pixelated particle detector implemented in high-voltage CMOS technology". In: *Nucl.Instrum.Meth.* A582 (2007), p. 876. DOI: `10.1016/j.nima.2007.07.115`.

[18] K. Olchanski S. Ritt P. Amaudruz. *Maximum Integration Data Acquisition System.* `http://midas.psi.ch|`. 2001. URL: `%5Cverb%7Chttp://midas.psi.ch%7C`.

[19] Goran Stanic. *A Camera Alignment System for the Mu3e Experiment*. 2021. URL: `https://www.psi.ch/en/media/69642/download?attachment`.

List of Figures

List of Tables

Appendix A

```
Packet ID,Time [s],Address,Read/Write,Reg_Addr,Data,ACK/NAK
0,1.700380125,0x10,Write,0x0100,0x00,ACK
1,1.700781875,0x10,Write,0x30EB,0x0C,ACK
2,1.70118291666667,0x10,Write,0x30EB,0x05,ACK
3,1.70158333333333,0x10,Write,0x300A,0xFF,ACK
4,1.701984,0x10,Write,0x300B,0xFF,ACK
5,1.70238429166667,0x10,Write,0x30EB,0x05,ACK
6,1.702784875,0x10,Write,0x30EB,0x09,ACK
7,1.703185625,0x10,Write,0x0114,0x01,ACK
8,1.70358583333333,0x10,Write,0x0128,0x00,ACK
9,1.70398616666667,0x10,Write,0x012A,0x18,ACK
10,1.704387125,0x10,Write,0x012B,0x00,ACK
11,1.70478745833333,0x10,Write,0x0164,0x00,ACK
12,1.70518970833333,0x10,Write,0x0165,0x00,ACK
13,1.70560554166667,0x10,Write,0x0166,0x0C,ACK
14,1.70600595833333,0x10,Write,0x0167,0xCF,ACK
15,1.706408,0x10,Write,0x0168,0x00,ACK
16,1.70680891666667,0x10,Write,0x0169,0x00,ACK
17,1.70720929166667,0x10,Write,0x016A,0x09,ACK
18,1.70761004166667,0x10,Write,0x016B,0x9F,ACK
19,1.70801095833333,0x10,Write,0x016C,0x06,ACK
20,1.708411625,0x10,Write,0x016D,0x68,ACK
21,1.70881191666667,0x10,Write,0x016E,0x04,ACK
22,1.70921225,0x10,Write,0x016F,0xD0,ACK
23,1.709613375,0x10,Write,0x0170,0x01,ACK
24,1.71001395833333,0x10,Write,0x0171,0x01,ACK
25,1.71041454166667,0x10,Write,0x0174,0x01,ACK
26,1.71081591666667,0x10,Write,0x0175,0x01,ACK
27,1.71121729166667,0x10,Write,0x0301,0x05,ACK
28,1.71162120833333,0x10,Write,0x0303,0x01,ACK
29,1.71202154166667,0x10,Write,0x0304,0x03,ACK
30,1.71242183333333,0x10,Write,0x0305,0x03,ACK
31,1.7128225,0x10,Write,0x0306,0x00,ACK
32,1.71322279166667,0x10,Write,0x0307,0x39,ACK
33,1.71362354166667,0x10,Write,0x030B,0x01,ACK
34,1.714023875,0x10,Write,0x030C,0x00,ACK
35,1.71442441666667,0x10,Write,0x030D,0x72,ACK
36,1.71482566666667,0x10,Write,0x0624,0x06,ACK
37,1.71522758333333,0x10,Write,0x0625,0x68,ACK
38,1.71563108333333,0x10,Write,0x0626,0x04,ACK
39,1.71603495833333,0x10,Write,0x0627,0xD0,ACK
40,1.71644475,0x10,Write,0x455E,0x00,ACK
41,1.71684854166667,0x10,Write,0x471E,0x4B,ACK
42,1.71724908333333,0x10,Write,0x4767,0x0F,ACK
43,1.71764941666667,0x10,Write,0x4750,0x14,ACK
44,1.71805154166667,0x10,Write,0x4540,0x00,ACK
45,1.71846275,0x10,Write,0x47B4,0x14,ACK
46,1.7188665,0x10,Write,0x4713,0x30,ACK
47,1.71926845833333,0x10,Write,0x478B,0x10,ACK
48,1.71967358333333,0x10,Write,0x478F,0x10,ACK
49,1.720077875,0x10,Write,0x4793,0x10,ACK
50,1.72047825,0x10,Write,0x4797,0x0E,ACK
51,1.72088016666667,0x10,Write,0x479B,0x0E,ACK
52,1.72128070833333,0x10,Write,0x0162,0x0D,ACK
53,1.721681625,0x10,Write,0x0163,0x78,ACK
54,1.72208541666667,0x10,Write,0x018C,0x0A,ACK
55,1.72248654166667,0x10,Write,0x018D,0x0A,ACK
56,1.72288866666667,0x10,Write,0x0309,0x0A,ACK
57,1.7233045,0x10,Write,0x0160,0x06E3,ACK
58,1.72379858333333,0x10,Write,0x015A,0x0034,ACK
59,1.72438204166667,0x10,Write,0x0157,0x00,ACK
60,1.72478641666667,0x10,Write,0x0158,0x0100,ACK
61,1.72528020833333,0x10,Write,0x0172,0x03,ACK
62,1.72568545833333,0x10,Write,0x0172,0x03,ACK
```

```
63,1.726087666666667,0x10,Write,0x0600,0x0000,ACK
64,1.726580291666667,0x10,Write,0x0602,0x03FF,ACK
65,1.727072541666667,0x10,Write,0x0604,0x03FF,ACK
66,1.727565416666667,0x10,Write,0x0606,0x03FF,ACK
67,1.728057208333333,0x10,Write,0x0608,0x03FF,ACK
68,1.728548791666667,0x10,Write,0x0100,0x01,ACK
69,1.832301083333333,0x10,Write,0x015A,0x06DF,ACK
70,1.865704958333333,0x10,Write,0x0157,0xE0,ACK
71,6.768608333333333,0x10,Write,0x0100,0x00,ACK
```

Listing 4.1: cleaned-up I²C start-up transaction data between a Raspberry Pi 3B and the IMX219 sensor module

Appendix B

```
--------------------------------------------------------------------------------
--
--   FileName:         i2c_master.vhd
--   Dependencies:     none
--   Design Software:  Quartus II 64-bit Version 13.1 Build 162 SJ Full Version
--
--   HDL CODE IS PROVIDED "AS IS."  DIGI-KEY EXPRESSLY DISCLAIMS ANY
--   WARRANTY OF ANY KIND, WHETHER EXPRESS OR IMPLIED, INCLUDING BUT NOT
--   LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A
--   PARTICULAR PURPOSE, OR NON-INFRINGEMENT. IN NO EVENT SHALL DIGI-KEY
--   BE LIABLE FOR ANY INCIDENTAL, SPECIAL, INDIRECT OR CONSEQUENTIAL
--   DAMAGES, LOST PROFITS OR LOST DATA, HARM TO YOUR EQUIPMENT, COST OF
--   PROCUREMENT OF SUBSTITUTE GOODS, TECHNOLOGY OR SERVICES, ANY CLAIMS
--   BY THIRD PARTIES (INCLUDING BUT NOT LIMITED TO ANY DEFENSE THEREOF),
--   ANY CLAIMS FOR INDEMNITY OR CONTRIBUTION, OR OTHER SIMILAR COSTS.
--
--   Version History
--   Version 1.0 11/01/2012 Scott Larson
--     Initial Public Release
--   Version 2.0 06/20/2014 Scott Larson
--     Added ability to interface with different slaves in the same transaction
--     Corrected ack_error bug where ack_error went 'Z' instead of '1' on error
--     Corrected timing of when ack_error signal clears
--   Version 2.1 10/21/2014 Scott Larson
--     Replaced gated clock with clock enable
--     Adjusted timing of SCL during start and stop conditions
--   Version 2.2 02/05/2015 Scott Larson
--     Corrected small SDA glitch introduced in version 2.1
--
--------------------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;


ENTITY i2c_master IS
  GENERIC(
    input_clk : INTEGER := 50_000_000; --input clock speed from user logic in Hz
    bus_clk   : INTEGER := 400_000);   --speed the i2c bus (scl) will run at in
    Hz
  PORT(
    clk       : IN     STD_LOGIC;                    --system clock
    reset_n   : IN     STD_LOGIC;                    --active low reset
    ena       : IN     STD_LOGIC;                    --latch in command
    addr      : IN     STD_LOGIC_VECTOR(6 DOWNTO 0); --address of target slave
    rw        : IN     STD_LOGIC;                    --'0' is write, '1' is read
    data_wr   : IN     STD_LOGIC_VECTOR(7 DOWNTO 0); --data to write to slave
    busy      : OUT    STD_LOGIC;                    --indicates transaction in
    progress
    data_rd   : OUT    STD_LOGIC_VECTOR(7 DOWNTO 0); --data read from slave
    ack_error : BUFFER STD_LOGIC;                    --flag if improper
    acknowledge from slave
    sda       : INOUT  STD_LOGIC;                    --serial data output of i2c
    bus
    scl       : INOUT  STD_LOGIC);                   --serial clock output of i2c
     bus
END i2c_master;

ARCHITECTURE logic OF i2c_master IS
  CONSTANT divider  :  INTEGER := (input_clk/bus_clk)/4; --number of clocks in
    1/4 cycle of scl
  TYPE machine IS(ready, start, command, slv_ack1, wr, rd, slv_ack2, mstr_ack,
    stop); --needed states
  SIGNAL state         : machine;                        --state machine
  SIGNAL data_clk      : STD_LOGIC;                      --data clock for sda
  SIGNAL data_clk_prev : STD_LOGIC;                      --data clock during
    previous system clock
  SIGNAL scl_clk       : STD_LOGIC;                      --constantly running
    internal scl
  SIGNAL scl_ena       : STD_LOGIC := '0';               --enables internal scl
    to output
  SIGNAL sda_int       : STD_LOGIC := '1';               --internal sda
  SIGNAL sda_ena_n     : STD_LOGIC;                      --enables internal sda
    to output
  SIGNAL addr_rw       : STD_LOGIC_VECTOR(7 DOWNTO 0);   --latched in address and
     read/write
  SIGNAL data_tx       : STD_LOGIC_VECTOR(7 DOWNTO 0);   --latched in data to
    write to slave
  SIGNAL data_rx       : STD_LOGIC_VECTOR(7 DOWNTO 0);   --data received from
    slave
  SIGNAL bit_cnt       : INTEGER RANGE 0 TO 7 := 7;      --tracks bit number in
    transaction
  SIGNAL stretch       : STD_LOGIC := '0';               --identifies if slave is
     stretching scl
BEGIN

  --generate the timing for the bus clock (scl_clk) and the data clock (data_clk)
  PROCESS(clk, reset_n)
    VARIABLE count  :  INTEGER RANGE 0 TO divider*4;  --timing for clock
    generation
  BEGIN
    IF(reset_n = '0') THEN                 --reset asserted
      stretch <= '0';
      count := 0;
    ELSIF(clk'EVENT AND clk = '1') THEN
      data_clk_prev <= data_clk;           --store previous value of data clock
      IF(count = divider*4-1) THEN         --end of timing cycle
        count := 0;                        --reset timer
      ELSIF(stretch = '0') THEN            --clock stretching from slave not
    detected
        count := count + 1;                --continue clock generation timing
      END IF;
      CASE count IS
        WHEN 0 TO divider-1 =>             --first 1/4 cycle of clocking
          scl_clk <= '0';
          data_clk <= '0';
        WHEN divider TO divider*2-1 =>     --second 1/4 cycle of clocking
          scl_clk <= '0';
          data_clk <= '1';
        WHEN divider*2 TO divider*3-1 =>  --third 1/4 cycle of clocking
          scl_clk <= '1';                  --release scl
          IF(scl = '0') THEN               --detect if slave is stretching clock
            stretch <= '1';
          ELSE
            stretch <= '0';
          END IF;
          data_clk <= '1';
        WHEN OTHERS =>                     --last 1/4 cycle of clocking
          scl_clk <= '1';
          data_clk <= '0';
      END CASE;
    END IF;
  END PROCESS;

  --state machine and writing to sda during scl low (data_clk rising edge)
  PROCESS(clk, reset_n)
  BEGIN
    IF(reset_n = '0') THEN                  --reset asserted
      state <= ready;                       --return to initial state
      busy <= '1';                          --indicate not available
```

```vhdl
      scl_ena <= '0';                      --sets scl high impedance
      sda_int <= '1';                      --sets sda high impedance
      ack_error <= '0';                    --clear acknowledge error flag
      bit_cnt <= 7;                        --restarts data bit counter
      data_rd <= "00000000";               --clear data read port
    ELSIF(clk'EVENT AND clk = '1') THEN
      IF(data_clk = '1' AND data_clk_prev = '0') THEN  --data clock rising edge
        CASE state IS
          WHEN ready =>                      --idle state
            IF(ena = '1') THEN               --transaction requested
              busy <= '1';                   --flag busy
              addr_rw <= addr & rw;          --collect requested slave address
    and command
              data_tx <= data_wr;            --collect requested data to write
              state <= start;                --go to start bit
            ELSE                             --remain idle
              busy <= '0';                   --unflag busy
              state <= ready;                --remain idle
            END IF;
          WHEN start =>                      --start bit of transaction
            busy <= '1';                     --resume busy if continuous mode
            sda_int <= addr_rw(bit_cnt);     --set first address bit to bus
            state <= command;                --go to command
          WHEN command =>                    --address and command byte of
    transaction
            IF(bit_cnt = 0) THEN             --command transmit finished
              sda_int <= '1';                --release sda for slave acknowledge
              bit_cnt <= 7;                  --reset bit counter for "byte"
    states
              state <= slv_ack1;             --go to slave acknowledge (command)
            ELSE                             --next clock cycle of command state
              bit_cnt <= bit_cnt - 1;        --keep track of transaction bits
              sda_int <= addr_rw(bit_cnt-1); --write address/command bit to bus
              state <= command;              --continue with command
            END IF;
          WHEN slv_ack1 =>                   --slave acknowledge bit (command)
            IF(addr_rw(0) = '0') THEN        --write command
              sda_int <= data_tx(bit_cnt);   --write first bit of data
              state <= wr;                   --go to write byte
            ELSE                             --read command
              sda_int <= '1';                --release sda from incoming data
              state <= rd;                   --go to read byte
            END IF;
          WHEN wr =>                         --write byte of transaction
            busy <= '1';                     --resume busy if continuous mode
            IF(bit_cnt = 0) THEN             --write byte transmit finished
              sda_int <= '1';                --release sda for slave acknowledge
              bit_cnt <= 7;                  --reset bit counter for "byte"
    states
              state <= slv_ack2;             --go to slave acknowledge (write)
            ELSE                             --next clock cycle of write state
              bit_cnt <= bit_cnt - 1;        --keep track of transaction bits
              sda_int <= data_tx(bit_cnt-1); --write next bit to bus
              state <= wr;                   --continue writing
            END IF;
          WHEN rd =>                         --read byte of transaction
            busy <= '1';                     --resume busy if continuous mode
            IF(bit_cnt = 0) THEN             --read byte receive finished
              IF(ena = '1' AND addr_rw = addr & rw) THEN  --continuing with
    another read at same address
                sda_int <= '0';              --acknowledge the byte has been
    received
              ELSE                           --stopping or continuing with a
    write
                sda_int <= '1';              --send a no-acknowledge (before stop
     or repeated start)
              END IF;
              bit_cnt <= 7;                  --reset bit counter for "byte" states
              data_rd <= data_rx;            --output received data
              state <= mstr_ack;             --go to master acknowledge
            ELSE                             --next clock cycle of read state
              bit_cnt <= bit_cnt - 1;        --keep track of transaction bits
              state <= rd;                   --continue reading
            END IF;
          WHEN slv_ack2 =>                   --slave acknowledge bit (write)
            IF(ena = '1') THEN               --continue transaction
              busy <= '0';                   --continue is accepted
              addr_rw <= addr & rw;          --collect requested slave address and command
              data_tx <= data_wr;            --collect requested data to write
              IF(addr_rw = addr & rw) THEN   --continue transaction with another write
                sda_int <= data_wr(bit_cnt); --write first bit of data
                state <= wr;                 --go to write byte
              ELSE                           --continue transaction with a read or new slave
                state <= start;              --go to repeated start
              END IF;
            ELSE                             --complete transaction
              state <= stop;                 --go to stop bit
            END IF;
          WHEN mstr_ack =>                   --master acknowledge bit after a read
            IF(ena = '1') THEN               --continue transaction
              busy <= '0';                   --continue is accepted and data received is available on bus
              addr_rw <= addr & rw;          --collect requested slave address and command
              data_tx <= data_wr;            --collect requested data to write
              IF(addr_rw = addr & rw) THEN   --continue transaction with another read
                sda_int <= '1';              --release sda from incoming data
                state <= rd;                 --go to read byte
              ELSE                           --continue transaction with a write or new slave
                state <= start;              --repeated start
              END IF;
            ELSE                             --complete transaction
              state <= stop;                 --go to stop bit
            END IF;
          WHEN stop =>                       --stop bit of transaction
            busy <= '0';                     --unflag busy
            state <= ready;                  --go to idle state
        END CASE;
      ELSIF(data_clk = '0' AND data_clk_prev = '1') THEN  --data clock falling edge
        CASE state IS
          WHEN start =>
            IF(scl_ena = '0') THEN                  --starting new transaction
              scl_ena <= '1';                       --enable scl output
              ack_error <= '0';                     --reset acknowledge error output
            END IF;
          WHEN slv_ack1 =>                          --receiving slave acknowledge (command)
            IF(sda /= '0' OR ack_error = '1') THEN  --no-acknowledge or previous no-acknowledge
              ack_error <= '1';                     --set error output if no-acknowledge
            END IF;
          WHEN rd =>                                --receiving slave data
            data_rx(bit_cnt) <= sda;                --receive current slave data bit
```

```
          WHEN slv_ack2 =>                         --receiving slave acknowledge
    (write)
            IF(sda /= '0' OR ack_error = '1') THEN  --no-acknowledge or previous
    no-acknowledge
              ack_error <= '1';                    --set error output if no-
    acknowledge
            END IF;
          WHEN stop =>
            scl_ena <= '0';                        --disable scl
          WHEN OTHERS =>
            NULL;
        END CASE;
      END IF;
    END IF;
  END PROCESS;

  --set sda output
  WITH state SELECT
    sda_ena_n <= data_clk_prev WHEN start,     --generate start condition
                 NOT data_clk_prev WHEN stop,  --generate stop condition
                 sda_int WHEN OTHERS;          --set to internal sda signal

  --set scl and sda outputs
  scl <= '0' WHEN (scl_ena = '1' AND scl_clk = '0') ELSE 'Z';
  sda <= '0' WHEN sda_ena_n = '0' ELSE 'Z';

END logic;
```

Listing 4.2: i²C controller by Scott Larson.[13]

```

----------------------------------------------------------
--      CCI Interface for Raspberry Pi Cam V2            --
-- Philipp Freundlieb pfreundl@students.uni-mainz.de     --
----------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.CCI_constants.all;

ENTITY CCI is
port (
    -- i_clk               : in  std_logic;
    -- i_reset_n           : in  std_logic;
    button1             : in std_logic;
  enable        : out std_logic := '0';

    --I2C
    SCL                 : inout std_logic;
    SDA                 : inout std_logic;

    --LED
    R_LED             : out std_logic;
    G_LED             : out std_logic;
    B_LED             : out std_logic;
    led_bank          : out std_logic_vector(3 downto 0);

    --FIFOs
    i_DATA              : in std_logic_vector(31 downto 0);
    o_DATA              : out std_logic_vector(31 downto 0);
    i_WR_ENA            : in std_logic;
    i_RD_ENA            : in std_logic;
    o_BUFF_EMPTY        : out std_logic;
    i_BUFF_FULL         : out std_logic;         -- flag signalling externally,
    that input FIFO is unavailable

    -- MIPI PHY lanes
    i_mipi_clk        : in std_logic;
    i_mipi_data_0     : in std_logic;
    i_mipi_data_1     : in std_logic;

    -- i_mipi_clk_n        : in std_logic;
    -- i_mipi_clk_p        : in std_logic;
    -- i_mipi_data_0_n     : in std_logic;
    -- i_mipi_data_0_p     : in std_logic;
    -- i_mipi_data_1_n     : in std_logic;
    -- i_mipi_data_1_p     : in std_logic;

    -- CSI2 bridge and ECC outputs
  o_parallel_pixels   : out std_logic_vector(9 downto 0)  ; -- 10 bit wide
    parallel pixel output
    o_pixel_clock           : out std_logic;
    o_frame_valid           : out std_logic;
    o_line_valid            : out std_logic;
  o_packet_header     : out std_logic_vector(23 downto 0) ; -- corrected csi2
    packet header for easier data management at the front end
  o_ecc_errors      : out std_logic_vector(2 downto 0)    -- error indicator
);

END ENTITY;

architecture CCI_behave of CCI is

    signal osc                   : std_logic;        -- onboard oscillator
    signal reset_n               : std_logic;        -- internal reset
    signal CLK                   : std_logic;        -- main clock

    -- i2c
    signal i2c_rw               : std_logic;
    signal i2c_ena              : std_logic;
    signal i2c_busy             : std_logic;
    signal i2c_busy_prev        : std_logic;
    signal i2c_addr             : std_logic_vector(6 downto 0);
    signal i2c_data_rd          : std_logic_vector(7 downto 0);
    signal i2c_data_wr          : std_logic_vector(7 downto 0);

    signal target_add            : std_logic_vector(6 downto 0);
    signal rw_bit               : std_logic;
    signal reg_add              : std_logic_vector(15 downto 0);
    signal data                 : std_logic_vector(7 downto 0);

    -- internal state machine and latched control signals
    signal instruction_ID       : integer := 0;
    signal frame_signal_index   : integer range 0 to 6  := 0;
    -- index for wich per frame signal shall be sent next
    signal wait_counter         : integer := 0;                     -- counter to
     make CCI state machine wait (see constants for timing)
    signal init_signal_index    : integer := 1;                     --
    signal init_counter         : integer               := 0;       -- timer for
    separating i2c instructions and setting up registers in between
    signal button1_prev         : std_logic;
    signal pause_clock          : std_logic := '0';
    signal enable_wire          : std_logic;
    signal streaming            : std_logic;        -- indicates weather the
    camera shall be sending images
    signal setup_complete       : std_logic := '0';

    -- internal fifo wires
    signal in_buff_empty        : std_logic;
    signal in_buff_full         : std_logic;
    signal in_buff_full_override: std_logic;        -- easy way to avoid write
    conflicts on fifo
    signal in_buff_data         : std_logic_vector(31 downto 0);

    signal in_buff_wr_enable    : std_logic;
    signal read_in_buff         : std_logic := '0';


    -- latching registers (maybe not necessary, test timings for eventual
    improvement)
    signal queue_target_add      : std_logic_vector(6 downto 0);
    signal queue_rw_bit         : std_logic;
    signal queue_reg_add        : std_logic_vector(15 downto 0);
    signal queue_data           : std_logic_vector(7 downto 0);

    signal out_buff_empty       : std_logic;
    signal out_buff_full        : std_logic;    -- TODO: implement safeguard if
    out buffer is overflowing or signal error
    signal reg_add_out          : std_logic_vector(15 downto 0);
    signal out_buff_data        : std_logic_vector(15 downto 0);
    signal write_out_buff       : std_logic := '0';

    -- CSI2 bridge outputs
  signal virtual_channel      : std_logic_vector(1 downto 0);
  signal data_type        : std_logic_vector(5 downto 0);
  signal word_count       : std_logic_vector(15 downto 0);
  signal error_code       : std_logic_vector(7 downto 0);
    -- signal frame_valid       : std_logic;
  -- signal line_valid        : std_logic;
    -- signal o_parallel_pixels   : std_logic_vector(10 downto 0);


    component OSCH
  port (
    stdby : in std_logic := '0';
    sedstdby : out std_logic;
    osc : out std_logic
  );
    end component;

  -- Verilog component from Lattice IP catalogue
    component MIPI_CSI2_Serial2Parallel_Bridge
    port (sensor_clk    : in    std_logic;
        rstn          : in    std_logic;
    DCK             : in    std_logic;
        CH0             : in    std_logic;
        CH1             : in    std_logic;
        CH2             : in    std_logic;
        CH3             : in    std_logic;
    pixclk_adj      : out    std_logic;
    pixdata         : out    std_logic_vector(9 downto 0);
    fv          : out    std_logic;
    lv          : out    std_logic;
    vc          : out    std_logic_vector(1 downto 0);
    dt          : out    std_logic_vector(5 downto 0);
    wc          : out    std_logic_vector(15 downto 0);
    ecc         : out    std_logic_vector(7 downto 0)
    );
    end component;

attribute NOM_FREQ : string;
attribute NOM_FREQ of e_osch : label is "24.18";

constant CLOCK_SPEED                 : integer := 24_180_000;
constant ENABLE_HOLD                 : integer := CLOCK_SPEED/1000;          --
    wait 1ms between setting enable signal and starting i2c transactions
constant ENABLE_FADE                 : integer := CLOCK_SPEED/4_000;         --
    wait for transactions to finish before turning off enable signal
constant FRAME_TIME                  : integer := CLOCK_SPEED/60 + 1;        --
    wait 1/60th of a second to send the next frame call
constant BUS_SPEED                   : integer := 100_000;
constant INIT_TIME                   : integer := 3500;                      --
    time between concurrent i2c transactions

begin

    B_LED <= i2c_busy;                      -- signal idle in blue
    R_LED <= not(i2c_busy and i2c_rw);      -- signal read in red
    G_LED <= (not i2c_busy) or i2c_rw;      -- signal write in green

    led_bank(0) <= not SDA;
    led_bank(1) <= not SCL;
    led_bank(2) <= button1;
    led_bank(3) <= not in_buff_empty;

    i_BUFF_FULL <= in_buff_full_override;
    o_BUFF_EMPTY <= out_buff_empty;

    enable <= enable_wire;

    e_osch: OSCH
        port map
      (stdby => open,
            osc => CLK);          -- for TESTING using modelsim: change to osc
    signal and setup clockdiv for probing CLK

  mipi_to_parallel_lattice: component MIPI_CSI2_Serial2Parallel_Bridge
    port map (
    sensor_clk  => open,    -- sensor clock is not needed (D-PHY clock is
    continuous)
    rstn        => reset_n,
    DCK       => i_mipi_clk,
    CH0       => i_mipi_data_0,
    CH1       => i_mipi_data_1,
    CH2       => open,    -- not connected (RPI camera V2 only breaks out 2 D-PHY data lanes)
    CH3       => open,    -- not connected (RPI camera V2 only breaks out 2 D-PHY data lanes)
    pixclk_adj  => o_pixel_clock,
    pixdata   => o_parallel_pixels,
    fv      => o_frame_valid,
    lv      => o_line_valid,
    vc      => virtual_channel,
    dt      => data_type,
    wc      => word_count,
    ecc     => error_code
    );

    reset_generator_1: entity work.rst_gen
        port map
      (CLK        => CLK,
      reset     => reset_n);

    I2C_clock_div : entity work.clkdiv
        port map
            (o_clk    => open,    -- for TESTING: connect to CLK and double input clock freq if CLK is probed on osc
            i_reset_n   => '1',
            i_clk     => osc);

    i2c_master: entity work.i2c_master
        generic map(
            input_clk   => CLOCK_SPEED,     --input clock speed from user logic in Hz
            bus_clk     => BUS_SPEED        --speed the i2c bus (scl) will run at in Hz
        )

        port map(
            clk         => CLK,
            reset_n     => reset_n,
            ena         => i2c_ena,
            addr        => i2c_addr,
            rw          => i2c_rw,
            data_wr     => i2c_data_wr,
            busy        => i2c_busy,
            data_rd     => i2c_data_rd,
            ack_error   => open,
            sda         => SDA,
            scl         => SCL
        );

    fifo_in: entity work.fifo
      port map (
        Data        => in_buff_data,    -- 31..25 target_add | 24 rw_bit | 23..8 reg_add_in | 7..0 data_in
        WrClock     => CLK,
        RdClock     => CLK,
        WrEn        => in_buff_wr_enable,
        RdEn        => read_in_buff,
        Reset       => not reset_n,
        RPReset     => open,
        Q(31 downto 25) => queue_target_add,
        Q(24)           => queue_rw_bit,
        Q(23 downto 8)  => queue_reg_add,
        Q(7 downto 0)   => queue_data,
        Empty       => in_buff_empty,
        Full        => in_buff_full,
        AlmostEmpty => open,
        AlmostFull  => open
      );

    fifo_out: entity work.fifo
      port map (
        Data(31 downto 16)  => reg_add_out,
        Data(15 downto 0)   => out_buff_data,
        WrClock     => CLK,
        RdClock     => CLK,
        WrEn        => write_out_buff,
        RdEn        => i_RD_ENA,
        Reset       => not reset_n,
        RPReset     => open,
        Q           => o_DATA,        -- 31..16 reg_add | 15..0 out_buff_data
        Empty       => out_buff_empty,
        Full        => out_buff_full,
        AlmostEmpty => open,
        AlmostFull  => open
      );

  ecc: entity work.ecc
    port map(i_clk            => CLK,
              i_reset_n           => reset_n,
              ECC_in            => error_code,
              VC_in       => virtual_channel,
              DT_in       => data_type,
              WC_in       => word_count,
              corrected_error   => o_ecc_errors(1),
              higher_order_error  => o_ecc_errors(2),
              no_error      => o_ecc_errors(0),
              packet_header_out => o_packet_header
  );


    process(CLK, reset_n)
    begin
        if(reset_n = '0') then
            i2c_ena             <= '0';
            i2c_rw              <= '1';
            i2c_busy_prev       <= '0';
            instruction_ID      <=  0;
      enable_wire       <= '0';
            read_in_buff        <= '0';
            write_out_buff      <= '0';
            init_signal_index        <=  0;
            setup_complete      <= '0';
            wait_counter        <=  0;
            init_counter        <=  0;
            frame_signal_index  <=  0;
            streaming           <= '0';
            out_buff_data       <= b"0000000_000000000";
        -- ################################################################
        elsif(rising_edge(CLK)) then
            if (init_signal_index <= SETUP_TRANS_MAX_ID) then      -- process
    that fills fifo with setup array automatically
                in_buff_full_override <= '1';
                if (pause_clock = '0') then
                    in_buff_wr_enable <= '0';
                    pause_clock <= '1';
                    in_buff_data <= START_STREAM_ARRAY(init_signal_index);
                elsif (pause_clock = '1') then
                    in_buff_wr_enable <= '1';
                    pause_clock <= '0';
                    init_signal_index <= init_signal_index + 1;
                end if;
```

```vhdl
            elsif (init_signal_index = SETUP_TRANS_MAX_ID + 1) then
                in_buff_wr_enable <= '0';
                pause_clock <= '0';
                init_signal_index <= init_signal_index + 1;
            -- #############################################################
            else
                -- edge detector for button1 input
                button1_prev <= button1;
                if (button1_prev = '1' and button1 = '0' and (in_buff_empty = '0' or setup_complete = '0')) then
                    if (streaming = '0') then
                        enable_wire <= '1';
                        wait_counter <= ENABLE_HOLD - INIT_TIME;    -- timer sets "enable" signal 1ms before transactions start
                    else
                        streaming <= '0';
                    end if;
                end if;
                -- wait counter for different waiting periods
                if (wait_counter > 0) then
                    if (wait_counter = 1) then
                        init_counter <= INIT_TIME;
                        if (streaming = '1') then
                            wait_counter <= FRAME_TIME;
                            frame_signal_index <= 0;    -- reset per frame array index for current frame
                        end if;
                    end if;
                    wait_counter <= wait_counter - 1;
                end if;
                -- init counter for fifo and register init before transaction start
                if(init_counter > 0) then
                    if(init_counter = 3) then       -- cant use 1 because register change is too slow
                        if (in_buff_empty = '0') then
                            read_in_buff <= '1';
                        end if;
                    elsif(init_counter = 2) then
                        instruction_ID <= 1;        -- latch start pulse from button1
                        read_in_buff <= '0';
                        -- if (streaming = '1' and frame_signal_index <= PER_FRAME_TRANS_MAX_ID) then
                        --     test        <= REPEAT_PER_FRAME_ARRAY(frame_signal_index);
                        -- end if;
                    elsif(init_counter = 1) then
                        if (streaming = '1' and frame_signal_index <= PER_FRAME_TRANS_MAX_ID) then
                            target_add   <= REPEAT_PER_FRAME_ARRAY(frame_signal_index)(31 downto 25);
                            rw_bit      <= REPEAT_PER_FRAME_ARRAY(frame_signal_index)(24);
                            reg_add     <= REPEAT_PER_FRAME_ARRAY(frame_signal_index)(23 downto 8);
                            data        <= REPEAT_PER_FRAME_ARRAY(frame_signal_index)(7 downto 0);
                        elsif (streaming = '0') then
                            read_in_buff <= '0';

                            target_add   <= queue_target_add;
                            rw_bit      <= queue_rw_bit;
                            reg_add     <= queue_reg_add;
                            data        <= queue_data;
                        end if;
                    end if;
                    init_counter <= init_counter - 1;
                end if;
                -- control streaming setting (maybe make dependent on 0100 x1 transaction)
                if (in_buff_empty = '0' or setup_complete = '0') then
                    streaming <= '0';
                else
                    streaming <= '1';
                    -- wait_counter <= ENABLE_HOLD;
                end if;
                -- check for changing busy signal to send next instruction
                i2c_busy_prev   <= i2c_busy;
                if(i2c_busy_prev = '0' and i2c_busy = '1' and (wait_counter = 0 or streaming = '1')) then
                    instruction_ID <= instruction_ID + 1;
                end if;
                case instruction_ID is
                when 0 =>
                        write_out_buff <= '0';  -- disable write signal one clock after enable (in case 'others')
                when 1 =>       -- set camera adress and write first byte of target register
                    i2c_ena <= '1';
                    i2c_addr <= target_add;
                    i2c_rw <= '0';
                    i2c_data_wr <= reg_add(15 downto 8);
                when 2 =>       -- write second register address byte
                    i2c_data_wr <= reg_add(7 downto 0);
                when 3 =>       -- read or write one byte
                    i2c_rw <= rw_bit;
                    if (rw_bit = '0') then
                        i2c_data_wr <= data;
                    else
                        reg_add_out  <= reg_add;
                        out_buff_data(15 downto 8)   <= i2c_data_rd;
                    end if;
                when others =>
                    -- if (instruction_ID = 4 and not (data(7 downto 0) = x"00")) then   -- detect larger messages
                    --     i2c_rw <= rw_bit;
                    --     if (rw_bit = '0') then
                    --         i2c_data_wr <= data(7 downto 0);
                    --     else
                    --         reg_add_out  <= reg_add;
                    --         out_buff_data(7 downto 0)   <= i2c_data_rd;
                    --     end if;
                    -- else
                        if (streaming = '1') then
                            if (frame_signal_index <= PER_FRAME_TRANS_MAX_ID) then
                                frame_signal_index <= frame_signal_index + 1;
                            else
                                frame_signal_index <= 0;
                            end if;
                        end if;
                        if (rw_bit = '1') then
                            out_buff_data(15 downto 8)   <= i2c_data_rd;
                            write_out_buff <= '1';  -- enable writing recieved data to fifo_out
                        end if;
                        i2c_ena <= '0';
                        instruction_ID <= 0;            -- go to empty state and wait for button1 input or frame timer
                        if (frame_signal_index < PER_FRAME_TRANS_MAX_ID) then   -- do not restart setup after last transaction
                            init_counter <= INIT_TIME;      -- wait to induce separate adressing
                        elsif (streaming = '1') then
                            init_counter <= 0;
                            wait_counter <= FRAME_TIME;          -- necessary to get into wait counter control (could be nicer though)
                        end if;
                        if (in_buff_empty = '1') then
                            -- enable_fade <= 24_180_000/4_000; --engage enable_fade counter
                            setup_complete <= '1';
                            in_buff_data <= i_DATA;
                            in_buff_wr_enable <= i_WR_ENA;
                            in_buff_full_override <= in_buff_full;  -- release signal flag for normal operation of the fifo
                        end if;
                    -- end if;
                end case;
            end if;
        end if;
    end process;

end architecture;
```

Listing 4.3: CCI broker.

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity rst_gen is
port  (CLK        : in  std_logic;
    reset     : out   std_logic := '1');
end entity;

architecture rst_gen_behave of rst_gen is

    signal cnt : integer range 0 to 50000 := 0;

begin
    process(CLK)
    begin
    if rising_edge(CLK) then
      if cnt = 25000 then
        reset <= '0';
        cnt <= cnt+1;
      elsif cnt = 49900 then
        reset <= '1';
      elsif cnt < 49900 then
        cnt <= cnt+1;
      end if;
    end if;
  end process;

end architecture;
```

Listing 4.4: Reset generator.

```vhdl
-- VHDL netlist generated by SCUBA Diamond (64-bit) 3.12.0.240.2
-- Module  Version: 5.8
--C:\lscc\diamond\3.12\ispfpga\bin\nt64\scuba.exe -w -n fifo -lang vhdl -synth
    synplify -bus_exp 7 -bb -arch xo3c00a -type ebfifo -depth 256 -width 32 -
    rwidth 32 -no_enable -pe 1 -pf 1

-- Thu Sep 15 15:31:28 2022

library IEEE;
use IEEE.std_logic_1164.all;
-- synopsys translate_off
library MACHXO3L;
use MACHXO3L.components.all;
-- synopsys translate_on

entity fifo is
    port (
        Data: in  std_logic_vector(31 downto 0);
        WrClock: in  std_logic;
        RdClock: in  std_logic;
        WrEn: in  std_logic;
        RdEn: in  std_logic;
        Reset: in  std_logic;
        RPReset: in  std_logic;
        Q: out  std_logic_vector(31 downto 0);
        Empty: out  std_logic;
        Full: out  std_logic;
        AlmostEmpty: out  std_logic;
        AlmostFull: out  std_logic);
end fifo;

architecture Structure of fifo is

    -- internal signal declarations
    signal scuba_vhi: std_logic;
    signal Empty_int: std_logic;
    signal Full_int: std_logic;
    signal scuba_vlo: std_logic;

    -- local component declarations
    component VHI
        port (Z: out  std_logic);
    end component;
    component VLO
        port (Z: out  std_logic);
    end component;
    component FIFO8KB
        generic (FULLPOINTER1 : in String; FULLPOINTER : in String;
                AFPOINTER1 : in String; AFPOINTER : in String;
                AEPOINTER1 : in String; AEPOINTER : in String;
                ASYNC_RESET_RELEASE : in String; RESETMODE : in String;
                GSR : in String; CSDECODE_R : in String;
                CSDECODE_W : in String; REGMODE : in String;
                DATA_WIDTH_R : in Integer; DATA_WIDTH_W : in Integer);
        port (DI0: in  std_logic; DI1: in  std_logic; DI2: in  std_logic;
            DI3: in  std_logic; DI4: in  std_logic; DI5: in  std_logic;
            DI6: in  std_logic; DI7: in  std_logic; DI8: in  std_logic;
            DI9: in  std_logic; DI10: in  std_logic; DI11: in  std_logic;
            DI12: in  std_logic; DI13: in  std_logic;
            DI14: in  std_logic; DI15: in  std_logic;
            DI16: in  std_logic; DI17: in  std_logic;
            CSW0: in  std_logic; CSW1: in  std_logic;
            CSR0: in  std_logic; CSR1: in  std_logic;
            FULLI: in  std_logic; EMPTYI: in  std_logic;
            WE: in  std_logic; RE: in  std_logic; ORE: in  std_logic;
            CLKW: in  std_logic; CLKR: in  std_logic; RST: in  std_logic;
            RPRST: in  std_logic; DO0: out  std_logic;
            DO1: out  std_logic; DO2: out  std_logic;
            DO3: out  std_logic; DO4: out  std_logic;
            DO5: out  std_logic; DO6: out  std_logic;
            DO7: out  std_logic; DO8: out  std_logic;
            DO9: out  std_logic; DO10: out  std_logic;
            DO11: out  std_logic; DO12: out  std_logic;
            DO13: out  std_logic; DO14: out  std_logic;
            DO15: out  std_logic; DO16: out  std_logic;
            DO17: out  std_logic; EF: out  std_logic;
            AEF: out  std_logic; AFF: out  std_logic; FF: out  std_logic);
    end component;
    attribute syn_keep : boolean;
    attribute NGD_DRC_MASK : integer;
    attribute NGD_DRC_MASK of Structure : architecture is 1;

begin
    -- component instantiation statements
    fifo_0_1: FIFO8KB
        generic map (FULLPOINTER1=> "0b00111111110000", FULLPOINTER=> "0
    b01000000000000",
        AFPOINTER1=> "0b00000000000000", AFPOINTER=> "0b00000000010000",
        AEPOINTER1=> "0b00000000100000", AEPOINTER=> "0b00000000010000",
        ASYNC_RESET_RELEASE=> "SYNC", GSR=> "DISABLED", RESETMODE=> "ASYNC",
        REGMODE=> "NOREG", CSDECODE_R=> "0b11", CSDECODE_W=> "0b11",
        DATA_WIDTH_R=>  18, DATA_WIDTH_W=>  18)
        port map (DI0=>Data(0), DI1=>Data(1), DI2=>Data(2), DI3=>Data(3),
            DI4=>Data(4), DI5=>Data(5), DI6=>Data(6), DI7=>Data(7),
            DI8=>Data(8), DI9=>Data(9), DI10=>Data(10), DI11=>Data(11),
            DI12=>Data(12), DI13=>Data(13), DI14=>Data(14),
            DI15=>Data(15), DI16=>Data(16), DI17=>Data(17),
            CSW0=>scuba_vhi, CSW1=>scuba_vhi, CSR0=>scuba_vhi,
            CSR1=>scuba_vhi, FULLI=>Full_int, EMPTYI=>Empty_int,
            WE=>WrEn, RE=>RdEn, ORE=>RdEn, CLKW=>WrClock, CLKR=>RdClock,
            RST=>Reset, RPRST=>RPReset, DO0=>Q(9), DO1=>Q(10),
            DO2=>Q(11), DO3=>Q(12), DO4=>Q(13), DO5=>Q(14), DO6=>Q(15),
            DO7=>Q(16), DO8=>Q(17), DO9=>Q(0), DO10=>Q(1), DO11=>Q(2),
            DO12=>Q(3), DO13=>Q(4), DO14=>Q(5), DO15=>Q(6), DO16=>Q(7),
            DO17=>Q(8), EF=>Empty_int, AEF=>AlmostEmpty, AFF=>AlmostFull,
            FF=>Full_int);

    scuba_vhi_inst: VHI
        port map (Z=>scuba_vhi);

    scuba_vlo_inst: VLO
        port map (Z=>scuba_vlo);

    fifo_1_0: FIFO8KB
        generic map (FULLPOINTER1=> "0b00000000000000", FULLPOINTER=> "0
    b11111111110000",
        AFPOINTER1=> "0b00000000000000", AFPOINTER=> "0b11111111110000",
        AEPOINTER1=> "0b00000000000000", AEPOINTER=> "0b11111111110000",
        ASYNC_RESET_RELEASE=> "SYNC", GSR=> "DISABLED", RESETMODE=> "ASYNC",
        REGMODE=> "NOREG", CSDECODE_R=> "0b11", CSDECODE_W=> "0b11",
        DATA_WIDTH_R=>  18, DATA_WIDTH_W=>  18)
        port map (DI0=>Data(18), DI1=>Data(19), DI2=>Data(20),
            DI3=>Data(21), DI4=>Data(22), DI5=>Data(23), DI6=>Data(24),
            DI7=>Data(25), DI8=>Data(26), DI9=>Data(27), DI10=>Data(28),
            DI11=>Data(29), DI12=>Data(30), DI13=>Data(31),
            DI14=>scuba_vlo, DI15=>scuba_vlo, DI16=>scuba_vlo,
            DI17=>scuba_vlo, CSW0=>scuba_vhi, CSW1=>scuba_vhi,
            CSR0=>scuba_vhi, CSR1=>scuba_vhi, FULLI=>Full_int,
            EMPTYI=>Empty_int, WE=>WrEn, RE=>RdEn, ORE=>RdEn,
            CLKW=>WrClock, CLKR=>RdClock, RST=>Reset, RPRST=>RPReset,
            DO0=>Q(27), DO1=>Q(28), DO2=>Q(29), DO3=>Q(30), DO4=>Q(31),
            DO5=>open, DO6=>open, DO7=>open, DO8=>open, DO9=>Q(18),
            DO10=>Q(19), DO11=>Q(20), DO12=>Q(21), DO13=>Q(22),
            DO14=>Q(23), DO15=>Q(24), DO16=>Q(25), DO17=>Q(26), EF=>open,
            AEF=>open, AFF=>open, FF=>open);

    Empty <= Empty_int;
    Full <= Full_int;
end Structure;

-- synopsys translate_off
library MACHXO3L;
configuration Structure_CON of fifo is
    for Structure
        for all:VHI use entity MACHXO3L.VHI(V); end for;
        for all:VLO use entity MACHXO3L.VLO(V); end for;
        for all:FIFO8KB use entity MACHXO3L.FIFO8KB(V); end for;
    end for;
end Structure_CON;

-- synopsys translate_on
```

Listing 4.5: Lattice FIFO.

```
library ieee;
use ieee.std_logic_1164.all;

-- clock divider
-- Period(o_clk) = 2 * N * Period(i_clk)
entity clkdiv is
generic (
    N : positive := 1--;
);
port (
    o_clk       : out   std_logic;
    i_reset_n   : in    std_logic := '1';
    i_clk       : in    std_logic--;
);
end entity;

```

```
architecture arch of clkdiv is

    signal clk : std_logic := '0';
    signal cnt : integer range 0 to N-1 := 0;

begin

    o_clk <= clk;

    process(i_clk, i_reset_n)
    begin
    if ( i_reset_n = '0' ) then
        clk <= '0';
        cnt <= 0;
        --
    elsif rising_edge(i_clk) then
        if ( cnt = N-1 ) then
            clk <= not clk;
            cnt <= 0;
        else
            cnt <= cnt + 1;
        end if;
        --
    end if;
    end process;

end architecture;
```

Listing 4.6: Clock divider.

```vhdl

----------------------------------------------------------
--       Error Correction Module for FPGA bridge         --
-- Philipp Freundlieb pfreundl@students.uni-mainz.de     --
----------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_misc.xor_reduce;


ENTITY ECC is
port (
    i_clk                : in std_logic;
    i_reset_n            : in std_logic;

  --packet header elements
  ECC_in          : in std_logic_vector(7 downto 0);    -- error correction code in the arriving packet
  VC_in         : in std_logic_vector(1 downto 0);    -- virtual channel identifier MSB of data identifier DI
  DT_in         : in std_logic_vector(5 downto 0);    -- data type LSB of data identifier
  WC_in         : in std_logic_vector(15 downto 0);   -- word count

  --error flags
  corrected_error     : out std_logic;    -- a single bit error has been found and corrected
  higher_order_error    : out std_logic;    -- two or more bits are erroneous
  no_error        : out std_logic;    -- no error has been detected

  packet_header_out   : out std_logic_vector(23 downto 0)   -- corrected packet header
);

END ENTITY;

architecture ECC_behave of ECC is

  type syndrome_array is array (0 to 23) of std_logic_vector(7 downto 0);

    --hard coded lookup matrix, taken from the CSI-2 documentation
      constant SYNDROM_LOOKUP   : syndrome_array :=
        (x"07", x"0B", x"0D", x"0E", x"13", x"15", x"16", x"19",
    x"1A", x"1C", x"23", x"25", x"26", x"29", x"2A", x"2C",
    x"31", x"32", x"34", x"38", x"1F", x"2F", X"37", x"3B");


  signal parity_in : std_logic_vector(7 downto 0);      -- transmitted parity byte
  signal header_data : std_logic_vector(23 downto 0);     -- recombined
  signal parity : std_logic_vector(7 downto 0);       -- newly calculated paritybits
  signal syndrome : std_logic_vector(7 downto 0);       -- syndrome of parity comparison
  signal single_header_error : std_logic;           -- flag wether a singular error in the header occured
  signal temp_packet_header : std_logic_vector(7 downto 0);

begin
  -- note: this process might need more pipelining to work properly!
    process(i_reset_n, i_clk)
        begin
      if i_reset_n = '0' then
        -- TODO reset here
      elsif rising_edge(i_clk) then
        parity_in <= ECC_in(7 downto 0);        -- latch parity byte
        header_data <= WC_in & VC_in & DT_in;       -- latch header data for rx ecc calculation

        -- bit vectors and formula taken from CSI-2 documentation
        parity <= "00" &
               xor_reduce(header_data xor "111011111111110000000000") &
               xor_reduce(header_data xor "110111110000001111110000") &
               xor_reduce(header_data xor "101110001110001110001110") &
               xor_reduce(header_data xor "011101001001101001101101") &
               xor_reduce(header_data xor "111100100101010101011011") &
               xor_reduce(header_data xor "111100010010110010110111");

        syndrome <= parity_in xor parity;

        if syndrome = "00000000" then
          -- parity in and calculated parity byte are the same -> no error
          no_error <= '1';
          corrected_error <= '0';
          higher_order_error <= '0';
        elsif (syndrome = "00000001" or syndrome = "00000010" or syndrome = "00000100" or syndrome = "00001000" or syndrome = "00010000" or syndrome = "00100000" or syndrome = "01000000" or syndrome = "10000000") then
          -- the incoming ecc byte is erroneous, but can be corrected using the syndrome itself:
          packet_header_out(7 downto 0) <= parity_in xor syndrome;
          no_error <= '0';
          corrected_error <= '1';
          higher_order_error <= '0';
        else
          single_header_error <= '0';   -- reset for later hook
          for index in 0 to 23 loop
            if syndrome = SYNDROM_LOOKUP(index) then
             header_data(index) <= not header_data(index);   -- check synthesis result for this, might get far larger than expected
             packet_header_out <= header_data;
             no_error <= '0';
             corrected_error <= '1';
             higher_order_error <= '0';
             single_header_error <= '1';
            end if;
          end loop;
          if single_header_error = '0' then -- hook (to keep else case out of for-generator)
          -- more than one bit error occured -> non-recoverable
            no_error <= '0';
            corrected_error <= '0';
            higher_order_error <= '1';
          end if;
        end if;
      end if;
        end process;
end architecture;
```

Listing 4.7: Error correction module.

```
// ---------------------------------------------------------------------
// >>>>>>>>>>>>>>>>>>>>>>>>> COPYRIGHT NOTICE <<<<<<<<<<<<<<<<<<<<<<<<<
// ---------------------------------------------------------------------
// Copyright (c) 2006 by Lattice Semiconductor Corporation
// ---------------------------------------------------------------------
//
// Permission:
//
//   Lattice Semiconductor grants permission to use this code for use
//   in synthesis for any Lattice programmable logic product.  Other
//   use of this code, including the selling or duplication of any
//   portion is strictly prohibited.
//
// Disclaimer:
//
//   This VHDL or Verilog source code is intended as a design reference
//   which illustrates how these types of functions can be implemented.
//   It is the user's responsibility to verify their design for
//   consistency and functionality through the use of formal
//   verification methods.  Lattice Semiconductor provides no warranty
//   regarding the use or functionality of this code.
//
// ---------------------------------------------------------------------
//
//                       Lattice Semiconductor Corporation
//                       5555 NE Moore Court
//                       Hillsboro, OR 97214
//                       U.S.A
//
//                       TEL: 1-800-Lattice (USA and Canada)
//                            408-826-6000 (other locations)
//
//                       web: http://www.latticesemi.com/
//                       email: techsupport@latticesemi.com
//
// ---------------------------------------------------------------------
//
// This is a serial DDR LVDS to SDR CMOS parallel bridge
//
//
// ---------------------------------------------------------------------
//
// Revision History :
// ---------------------------------------------------------------------
//   Ver  :| Author            :| Mod. Date :| Changes Made:
//   V1.0 :| GJenning          :| 05/09/12  :| Alpha Release
//
    ------------------------------------------------------------------------------------------------------------------

module MIPI_CSI2_Serial2Parallel_Bridge #(
   parameter bus_width          = 10        ,   // 6-24             - the width of
     the pixel data, only formats supported by CSI2 Specification
   parameter lane_width         = 2         ,   // 1, 2, 4          - the number
    of lanes used, 10 bit or 12 bit widths supported
   parameter lp_mode            = "OFF"      ,   // OFF, ON          - changes
    clocking scheme for use when clock contains both HS and LP modes; OFF = free
    running clock, ON=LP mode in blanking periods
   parameter line_length_detect = 0         ,   // 0, 1             - setting this
     to one will only allow line valid to go active for lines of size line_length
   parameter line_length        = 2400      ,   //                  - Only valid
    if line_length_detect = 1.  Number of bytes in a line.  Example: 1920, 10 bit
     pixels per line = 1920*10/8 = 2400.  Should be set to the expected value of
    word count in the CSI2 Packet Header
   parameter format             = "RAW10"     )   //                  - Defines
    output format type
(
   input  wire                      rstn        ,   // reset, active low
   input  wire                      DCK         ,   // serial input clock

   input                            CH0     ,
   input                            CH1     ,
   input                            CH2     ,
   input                            CH3     ,   // LVDS DDR input data
   output                           pixclk_adj  ,   // output pixel clock
   output [bus_width-1:0]           pixdata     ,   // output SDR data(LVCMOS)
   output                           fv          ,   // frame valid output
   output                           lv          ,   // line valid output
   output [1:0]                     vc          ,   // virtual channel identifier output
   output [5:0]                     dt          ,   // data type output
   output [15:0]                    wc          ,   // word count output
   output [7:0]                     ecc         ,   // error correction code output

   input                            sensor_clk

);

wire [31:0] din, ch_din;
wire pixclk;

deser deser(.alignwd(1'b0), .clk(DCK), .clk_s(sensor_clk), .init(1'b1), .reset(~rstn), .rx_ready(rx_ready), .sclk(sclk_in), /* .clk_div2(clk_div2),*/ .datain({CH3,CH2,CH1,CH0}), .q(din));

MIPI_CSI2_Serial2Parallel #(.bus_width(bus_width), .lane_width(lane_width), .format(format)) serial2parallel(.rstn(lock), .din(ch_din), .fv(fv), .lv(lv), .pixdata(pixdata), .pixclk(pixclk), .sclk(sclk), .vc(vc), .dt(dt), .wc(wc), .ecc(ecc), .line_length_detect(line_length_detect), .line_length(line_length));

generate
     if      (bus_width==8  & lane_width == 1 & lp_mode=="ON")
          pll_8bit_1lane_lp pll(.CLKI(sensor_clk), .RST(~rstn), .CLKOP(pixclk), .CLKOS(pixclk_adj), .CLKOS2(sclk), .LOCK(lock));
     else if (bus_width==8  & lane_width == 2 & lp_mode=="ON")
          pll_8bit_2lane_lp pll(.CLKI(sensor_clk), .RST(~rstn), .CLKOP(pixclk), .CLKOS(pixclk_adj), .CLKOS2(sclk), .LOCK(lock));
     else if (bus_width==10 & lane_width == 4 & lp_mode=="ON")
          pll_10bit_4lane_lp pll(.CLKI(sensor_clk), .RST(~rstn), .CLKOP(pixclk), .CLKOS(pixclk_adj), .CLKOS2(sclk), .LOCK(lock));
     else if (bus_width==12 & lane_width == 4 & lp_mode=="ON")
          pll_12bit_4lane_lp pll(.CLKI(sensor_clk), .RST(~rstn), .CLKOP(pixclk), .CLKOS(pixclk_adj), .CLKOS2(sclk),  .LOCK(lock));
     else if (bus_width==14 & lane_width == 4 & lp_mode=="ON")
          pll_14bit_4lane_lp pll(.CLKI(sensor_clk), .RST(~rstn), .CLKOP(pixclk), .CLKOS(pixclk_adj), .CLKOS2(sclk),  .LOCK(lock));
     else if (bus_width==10 & lane_width == 2 & lp_mode=="ON")
          pll_10bit_2lane_lp pll(.CLKI(sensor_clk), .RST(~rstn), .CLKOP(pixclk), .CLKOS(pixclk_adj), .CLKOS2(sclk),  .LOCK(lock));
     else if (bus_width==12 & lane_width == 2 & lp_mode=="ON")
          pll_12bit_2lane_lp pll(.CLKI(sensor_clk), .RST(~rstn), .CLKOP(pixclk), .CLKOS(pixclk_adj), .CLKOS2(sclk),  .LOCK(lock));
     else if (bus_width==10 & lane_width == 1 & lp_mode=="ON")
          pll_10bit_1lane_lp pll(.CLKI(sensor_clk), .RST(~rstn), .CLKOP(pixclk), .CLKOS(pixclk_adj), .CLKOS2(sclk),  .LOCK(lock));
     else if (bus_width==12 & lane_width == 1 & lp_mode=="ON")
          pll_12bit_1lane_lp pll(.CLKI(sensor_clk), .RST(~rstn), .CLKOP(pixclk), .CLKOS(pixclk_adj), .CLKOS2(sclk),  .LOCK(lock));
     else if (bus_width==8  & lane_width == 1 & lp_mode=="OFF") begin
          pll_8bit_1lane pll(.CLKI(sclk_in), .RST(~rstn), .CLKOP(pixclk), .CLKOS(pixclk_adj), .LOCK(lock));  assign sclk = sclk_in; end
     else if (bus_width==8  & lane_width == 2 & lp_mode=="OFF") begin
```

```
          pll_8bit_2lane pll(.CLKI(sclk_in), .RST(~rstn), .CLKOP(pixclk), .CLKOS(pixclk_adj), .LOCK(lock));  assign sclk = sclk_in; end
   else if (bus_width==10 & lane_width == 4 & lp_mode=="OFF") begin
          pll_10bit_4lane pll(.CLKI(sclk_in), .RST(~rstn), .CLKOP(pixclk), .CLKOS(pixclk_adj), .LOCK(lock));  assign sclk = sclk_in; end
     else if (bus_width==12 & lane_width == 4 & lp_mode=="OFF") begin
          pll_12bit_4lane pll(.CLKI(sclk_in), .RST(~rstn), .CLKOP(pixclk), .CLKOS(pixclk_adj), .LOCK(lock));  assign sclk = sclk_in; end
     else if (bus_width==10 & lane_width == 2 & lp_mode=="OFF") begin
          pll_10bit_2lane pll(.CLKI(sclk_in), .RST(~rstn), .CLKOP(pixclk), .CLKOS(pixclk_adj), .LOCK(lock));  assign sclk = sclk_in; end
     else if (bus_width==12 & lane_width == 2 & lp_mode=="OFF") begin
          pll_12bit_2lane pll(.CLKI(sclk_in), .RST(~rstn), .CLKOP(pixclk), .CLKOS(pixclk_adj), .LOCK(lock));  assign sclk = sclk_in; end
     else if (bus_width==10 & lane_width == 1 & lp_mode=="OFF") begin
          pll_10bit_1lane pll(.CLKI(sclk_in), .RST(~rstn), .CLKOP(pixclk), .CLKOS(pixclk_adj), .LOCK(lock));  assign sclk = sclk_in; end
     else if (bus_width==12 & lane_width == 1 & lp_mode=="OFF") begin
          pll_12bit_1lane pll(.CLKI(sclk_in), .RST(~rstn), .CLKOP(pixclk), .CLKOS(pixclk_adj), .LOCK(lock));  assign sclk = sclk_in; end
     else if (bus_width==14 & lane_width == 4 & lp_mode=="OFF") begin
          pll_14bit_4lane pll(.CLKI(clk_div2), .RST(~rstn), .CLKOP(pixclk), .CLKOS(pixclk_adj), .LOCK(lock));  assign sclk = sclk_in; end
     else if (bus_width==18 & lane_width == 4 & lp_mode=="OFF") begin
          pll_18bit_4lane pll(.CLKI(sclk_in), .RST(~rstn), .CLKOP(pixclk), .CLKOS(pixclk_adj), .LOCK(lock));  assign sclk = sclk_in; end
     else if (bus_width==24 & lane_width == 4 & lp_mode=="OFF") begin
          pll_24bit_4lane pll(.CLKI(sclk_in), .RST(~rstn), .CLKOP(pixclk), .CLKOS(pixclk_adj), .LOCK(lock));  assign sclk = sclk_in; end
endgenerate

generate
     if       (lp_mode=="ON")
          LP_crossclk_cnvrt LP_crossclk_cnvrt(.Data(din), .WrClock(sclk_in), .RdClock(sclk), .WrEn(1), .RdEn(1), .Reset(~rstn), .RPReset(~rstn), .Q(ch_din), .Empty(deser_ready));
     else if  (lp_mode=="OFF")
          assign ch_din = din;
endgenerate

endmodule
```

Listing 4.8: Lattice CSI-2 bridge main component.