UPTEC F 23026
Examensarbete 30 hp
06 23

Modeling, Simulation, and Injection of Camera Images/Video to Automotive Embedded ECU

Image Injection Solution for Hardware-in-the-Loop Testing

Anton Lind

Civilingenjörsprogrammet i teknisk fysik

Modeling, Simulation and Injection of Camera Images/Video to Automotive Embedded ECU

Anton Lind

Abstract

Testing, verification and validation of sensors, components and systems is vital in the early-stage development of new cars with computer-in-the-car architecture. This can be done with the help of the existing technique, hardware-in-the-loop (HIL) testing which, in the close loop testing case, consists of four main parts: Real-Time Simulation Platform, Sensor Simulation PC, Interface Unit (IU), and unit under test which is, for instance, a Vehicle Computing Unit (VCU).

The purpose of this degree project is to research and develop a proof of concept for in-house development of an image injection solution (IIS) on the IU in the HIL testing environment. A proof of concept could confirm that editing, customizing, and having full control of the IU is a possibility. This project was initiated by Volvo Cars to optimize the use of the HIL testing environment currently available, making the environment more changeable and controllable while the IIS remains a static system.

The IU is an MPSoC/FPGA based design that uses primarily Xilinx hardware and software (Vivado/Vitis) to achieve the necessary requirements for image injection in the HIL testing environment. It consists of three stages in series: input, image processing, and output.

The whole project was divided in three parts based on the three stages and carried out at Volvo Cars in cooperation by three students, respectively. The author of this thesis was responsible for the output stage, where the main goal was to find a solution for converting, preferably, AXI4 RAW12 image data into data on CSI2 format. This CSI2 data can then be used as input to serializers, which in turn transmit the data via fiber-optic cable on GMSL2 format to the VCU.

Associated with the output stage, extensive simulations and hardware tests have been done on a preliminary solution that partially worked on the hardware, producing signals in parts of the design that could be read and analyzed.

However, a final definite solution that fully functions on the hardware has not been found, because the work is at the initial phase of an advanced and very complex project. Presented in this thesis is: important theory regarding, for example, protocols CSI2, AXI4, GMSL2, etc., appropriate hardware selection for an IIS in HIL (FPGA, MPSoC, FMC, etc.), simulations of AXI4 and CSI2 signals, comparisons of those simulations with the hardware signals of an implemented design, and more. The outcome was heavily dependent on getting a certain hardware (TEF0010) to transmit the GMSL2 data. Since the wrong card was provided, this was the main problem that hindered the thesis from reaching a fully functioning implementation. However, these results provide a solid foundation for future work related to image injection in a HIL environment.

Teknisk-naturvetenskapliga fakulteten

Uppsala universitet, Utgivningsort Uppsala/Visby

Handledare: Rubén Calvillo Portales Ämnesgranskare: Ping Wu

Examinator: Tomas Nyberg

Populärvetenskaplig sammanfattning

Med den växande digitalisering av vårt samhälle har integrerad teknologin ökat markant inom vissa industrier. Detta är speciellt märkbart inom fordonsindustrin där bilar har gått från att vara rent bränsledrivna maskiner med några enstaka simpla elektriska funktioner, till att ha flera sammankopplade elektriska kommunikationsnätverk, bestående av flertal datorer som styr många funktioner av varierande komplexitet. Följaktligen, har fokuset för tillverkningen av bilar skiftat mot huvudsaklig smarta funktioner och lösningar som inte bara gör körningen mer behaglig för föraren, men också mer säker.

Ett exempel på sådana smarta funktioner är aktiva säkerhetsfunktioner, som försöker undvika att kollisioner uppstår. Dessa aktiva säkerhetsfunktioner faller vanligtvis under två olika kategorier: avancerade körhjälpssystem (ADAS) och automatiserade körsystem (AD eller ADS). ADAS omfattar funktioner som hjälper föraren till mer säker körning, som till exempel kollisionsundvikning, filhållningshjälp, dödavinkeldetektering och adaptiv farthållare, men föraren förblir den huvudsakliga kontrollanten av fordonet. Medan AD är automatiserad körning helt utan, eller nästintill utan, interaktion från föraren.

Eftersom AD och ADAS introducerar mycket komplex och specifik programvara som så småningom kommer att användas för fordon i trafik, krävs omfattande tester för att säkerställa att hårdvaran och programvaran fungerar tillsammans på ett tillförlitligt sätt. Det är här som testning blir en viktig del av utvecklingen av nya bilar med dator-i-bilen-arkitektur. Bland de befintliga testteknikerna ger hardware-in-the-loop (HIL) testning en testmiljö som används för att verifiera och validera komponenter och system i de tidiga utvecklingsstadierna. På så sätt kan verkliga elektriska styrenheter (ECUs) och fordonsdatorenheter (VCUs) testas på ett kontrollerat och repeterbart sätt. Detta kan uppnås genom att stimulera ECUs/VCUs med antingen verkliga inspelade data eller syntetiska data. Ett exempel på detta kan vara bildinjektion, där bilder injiceras i ECUn/VCUn med antingen inspelade kameradata som verklig data eller simulerade bild-data som syntetiska data.

Syftet med detta examensarbete är att undersöka och utveckla ett konceptbevis för intern utveckling av en bildinjektionslösning, anpassad för en HIL testmiljö. Detta konceptbevis skulle kunna bekräfta att redigering, anpassning och full kontroll över bildinjektionslösningen är en möjlighet. Eftersom Volvo Cars inte har någon kontroll över programvaran i den nuvarande lösningen, skulle detta kunna minska de förseningar som uppstår när oförutsedda ändringar görs i testmiljön. I grund och botten blir testmiljön mer föränderlig och kontrollerbar, trots att bildinjektionslösningen förblir ett statiskt system.

Denna bildinjektionslösning kallas Interface Unit (IU) och är en Field Programmable Array (FPGA) baserad design som främst använder Xilinx hårdvara och mjukvara (Vivado/Vitis) för att uppnå de nödvändiga kraven för bildinjektion i en HIL-testmiljö. Examensarbetet utfördes i samarbete mellan tre studenter. Denna rapport täcker dock slutsteget av IU:n, där huvudmålet är att hitta en lösning för att konvertera, helst AXI4 RAW12 bild-data till data på CSI2-format. Denna CSI2-data kan sedan användas för att överföra informationen till en VCU.

I slutändan hittades ingen slutgiltig lösning som fungerade fullt på hårdvaran. Detta tros vara ett problem med initialisering i vissa delar av designen. Omfattande simuleringar och hårdvarutester gjordes dock på en preliminär lösning, som delvis fungerade på hårdvaran och producerade signaler i delar av designen som kunde läsas och analyseras i verklig tid. Även om ingen slutgiltig lösning hittades, ger denna rapport ändå en grundlig förklaring av teori, teknisk bakgrund, hårdvarukrav, hårdvarukonfiguration, programvarukonfiguration, simuleringar, hårdvaruresultat, osv, för HIL bildinjektion. Vilket följaktligen gör det till en solid grund för framtida arbete inom området.

Acknowledgements

This thesis was carried out at Uppsala University in cooperation with Volvo Cars Gothenburg.

First and foremost I would like to present my huge appreciation to team manager Siddhant Gupta and supervisor Rubén Calvillo Portales at Volvo Cars, who provided guidance, inspiration and assistance, not only on a professional but also a personal level. Also, thank you to Ping Wu at Uppsala University who guided me through this thesis with useful feedback and information.

A big thank you to my thesis partners Justus Hoffman and Nidhi Chakrabhavi Basavaraju who provided a great cooperative environment which, through discussions and assistance, furthered the work we accomplished together.

Another huge thanks to the rest of the workers in the Volvo Cars HIL-team who made the work and the life outside of work way more enjoyable.

Last but not least, thank you to my family who are always there to support me in any endeavours.

List of Abbreviations

AD - Automated Driving Systems

ADAS - Advanced Driving Assistance Systems

CAN - Controller Area Network

CSI2 - Camera Serial Interface 2

DP - DisplayPort

ECU- Electronic Control Unit

FMC - FPGA Mezzanine Card

FPGA - Field Programmable Gate Array

GMSL2 - Gigabit Multimedia Serial Link 2

HIL - Hardware-in-the-Loop

IC - Integrated Circuit

IDE - Integrated Development Environment

IP - Intellectual Property

IU - Interface Unit

LIN - Local Interconnect Network

MPSoC - Multiprocessor System on a Chip

PL -Programmable Logic

PS- Processing System

RTL - Register Transfer Level

TB - Testbench

TRM - Technical Reference Manual

VCU - Vehicle Computing Unit

VIU - Vehicle Interface Unit

WFW - Waveform Window

Contents

1 Introduction

1.1 Background

When cars were starting to be produced the main focus was allowing people to travel longer distances in a shorter time than what had been possible before. Safety was not the main priority, however as cars became more advanced so did the safety features. Today keeping the costumers safe and protected is a central part of the manufacturing process in the automotive industry, since it is vital to build a brand that represents quality and safety. This signifies what Volvo stands for, as they have been actively working for safer automotive transportation since at least 1959 when Volvo engineer Nils Bolin invented the three-point seat belt and Volvo Cars waived the patent rights, allowing everyone to benefit from the invention [1].

When it comes to the automotive industry, the safety features implemented can be divided into two groups: passive and active safety. Passive safety features are those that help keep the driver safe by reacting to the collision in the event of an incident. An example of this can be the three-point seat belt or collision activated airbags, which both help keep the driver and passengers safe in the event of a collision. Meanwhile, active safety features are those that try to avoid any collisions occurring in the first place [2]. These active safety features usually fall under two different categories: Advanced Driving Assistance Systems (ADAS) and Automated Driving Systems (AD or ADS). The ADAS includes features which assist the driver with smart functionalities such as collision avoidance, lane keeping aid, blind spot detection, and adaptive cruise control, but the driver remains the main controller of the vehicle. While the AD is fully automated driving with no, or almost no, interaction from the driver. The software for these ADAS and AD features usually use sensors such as cameras and LIDARs as inputs [3][4].

Volvo's safety vision from year 2007 was to strive for zero serious injuries or deaths in Volvo manufactured vehicles. This mainly came down to passive safety features, which helped keep the driver safe in the event of a collision. However, today a new safety vision has been implemented, where the goal is to have zero cars produced by Volvo in any accidents altogether [5]. This is where active safety features come in handy, and development in these areas is crucial to reach the vision of zero collisions for Volvo cars.

Since AD and ADAS introduce highly complex and specific software that will eventually be used for vehicles in traffic, extensive testing is needed to ensure that the sensors and software work together in a reliable way. This is where testing becomes a vital part of the development of new cars with computer-in-the-car architecture. Among the existing testing techniques, hardware-in-the-loop (HIL) testing allows the user to verify sensors and validate components and systems in the early stages of development. With sensor simulation techniques, validation of autonomous driving throughout all stages of development becomes a possibility.

1.2 Overview of the Hardware-in-the-Loop Testing Environment

HIL is a testing environment that is used to verify and validate components and systems in the early stages of development. By doing so, real electrical control units (ECUs) and vehicle computing units (VCUs) can be tested in a controlled and repeatable way. This can be achieved by stimulating the ECUs with either real recorded data or synthetic data. One example of this can be image injection, where images are injected into the ECU with either recorded camera data as real data or simulated image data as synthetic data. In Figure 1.1 an example setup is seen that allows both open-loop HIL simulations with raw data input from a front camera, as well as closed-loop HIL simulations with synthetic data from the Sensor Simulation PC. Note that the toggle switch connected to the ECU is used to switch between the open-loop and the closed-loop HIL simulations. In this thesis the front looking camera, including imager and lens, is instead replaced and simulated by the HIL environment. This is the general setup of the HIL environment that the thesis will be based on. Though, most of the thesis will focus on the Interface Unit and how to bridge the gap between the Sensor Simulator PC and the VCU/ECU in a suitable and potentially improved way.

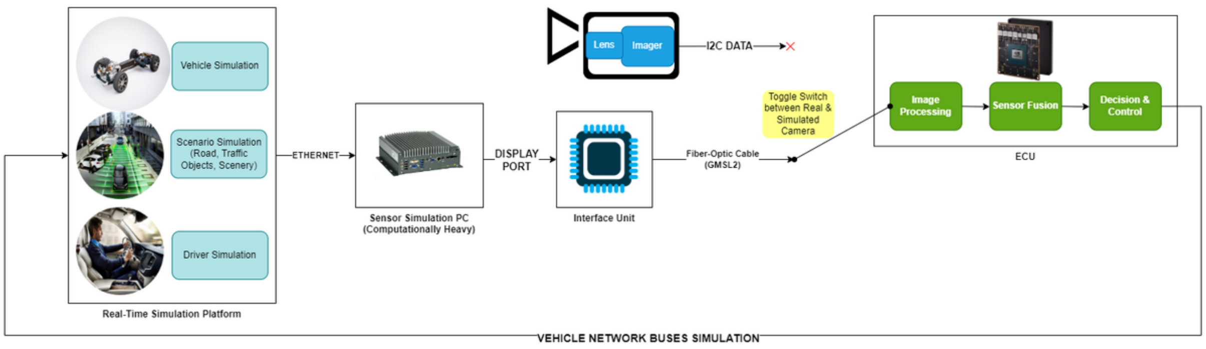

Vehicle Simulation
Scenario Simulation (Road, Traffic Objects, Scenery)
Driver Simulation
Real-Time Simulation Platform
ETHERNET
Sensor Simulation PC (Computationally Heavy)
DISPLAY PORT
Interface Unit
Lens
Imager
I2C DATA
Toggle Switch between Real & Simulated Camera
Fiber-Optic Cable (GMSL2)
Image Processing
Sensor Fusion
Decision & Control
ECU
VEHICLE NETWORK BUSES SIMULATION

Figure 1.1: HIL environment setup

To the left in Figure 1.1 is the *Real-Time Simulation Platform.* This is the simulation platform/software that generates the real-time 3D environment that is used during HIL testing. The software used in this case is called Aurelion. Through Aurelion, the 3D environment needed for image injection can be simulated from the exact perspective that the sensors would perceive the environment. This means, for example, that a certain camera can be simulated from its specific perspective, effectively seeing what it sees. An example of this is shown in Figure 1.2, where the perspective of a front looking camera in a car is simulated. This allows the testing environment to verify and validate specific sensors in the most realistic way.

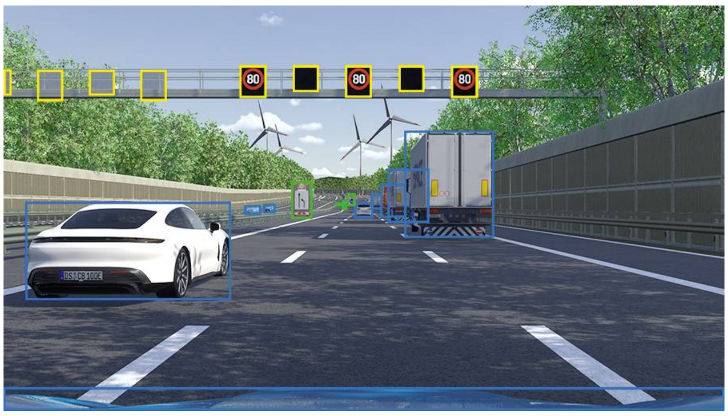


Figure 1.2: Camera perspective from Aurelion simulated 3D environment

However, this requires a substantial amount of graphical computational power. Hence, the platform runs on a *Sensor Simulation PC*, connected via Ethernet, that has multiple graphic cards that can handle the computationally heavy operations. The Sensor Simulation PC outputs the simulated images into DisplayPort format, which go directly to the *Interface Unit* (IU) as an input. In turn, the IU should output the image data in GMLS2 format via fiber-optic cable. This is then directly inserted into the VCU's image processing stack. The VCU processes the image information and feeds back commands/actions to the Real-Time Simulation Platform that the VCU wants to perform on the simulated vehicle. This HIL setup is thereby a closed-loop system. Note that the ECU in Figure 1.1 is a VCU in this thesis. Though, the same setup can be used with an ECU.

1.3 Scope of the Project

The goal of the thesis is to evaluate and implement the injection of the simulated (raw) image data directly into the ECU, by research and implementation in several areas as mentioned below:

1. Since the optical path generated from the camera, consisting of a lens and the imager, is removed; an emulation of these components is necessary in the visualization.

2. The I^{2}C data needs to be embedded
3. Timing and embedding of data that does not contain the actual image information need to be located and can thus be adapted and parameterized depending on the camera used.

Whether an FPGA or a Microcontroller is best suited for this objective and how they should be programmed to achieve this is the main research scope in this thesis. Emulation or Complete Emulation of 1) and 2) can be done partially on the GPU and partially on Microcontroller/FPGA based Interface Unit, leading to the second research area. For 3), it is evaluated to be best embedded on the Interface Unit, but that is another research scope in this project.

4. The connection between Sensor Simulation PC and the Interface Unit is facilitated via the Display Port output or any other viable interface. It is necessary to enable low-latency and efficient transmission of image data including reliability on the image quality with precise timing. This high precision timing enables limited data buffering resulting in minimizing the lag caused by signal adjustment. Selection of this interface is another research aspect.
5. Evaluation of this prototype system shall be carried out for raw camera data injection at an approximate rate of 10 Gbit/sec and the possibility to support multiple channels of data streams.

The original scope of the thesis, presented above, proved to be significantly broader and more complicated than anticipated as the work progressed. Due to this, the tasks and goals of the thesis as a whole had to be reconsidered and reduced in complexity, so that they could actually be carried out and achieved.

The focus of the whole thesis project will primarily be on how to implement the IU so that it is easily customizable while retaining sufficient performance. Today, there are some available solution for the IU, but the software is not publically available. So, as soon as something changes on either side of the IU (input/output), changes have to be made inside the IU that can not currently be done by Volvo. This causes long delays. The IU is also programmed with a bitstream using an SD card. This bitstream can not be reverse engineered, so it is impossible to know what code is being loaded onto the IU. Hence, the problem of how to program the IU has to be solved from the ground up.

The main goal stayed centered around the Interface Unit, but rather than implementing areas such as I^{2}C data, timing, embedding of data, automation, etc., the practical focus shifted more towards achieving a simple proof of concept. Partly, because some of these areas proved way too complicated, but also because none of them were really relevant until a basic solution had been found. To research and develop a simple solution for the IU, or parts of the IU, that can function in a simple environment became the new focus. Solely to prove that the IU can be edited, configured, fully controlled and eventually produced in-house was the new overarching goal for the thesis.

This specific thesis project and report will mainly focus on how to program the output stage of the IU. In particular, most of the work will be focused on how to convert, preferably RAW12, image data received in AXI4 format into MIPI CSI2 format. A more detailed block diagram of the system that the thesis focuses on is shown in Figure 1.3. The dash-line arrow for the GMSL2 connection represents how the camera would usually be connected to the VCU. However, the camera is replaced by the Simulator and the IU that together produce the input for the VCU (i.e image injection).

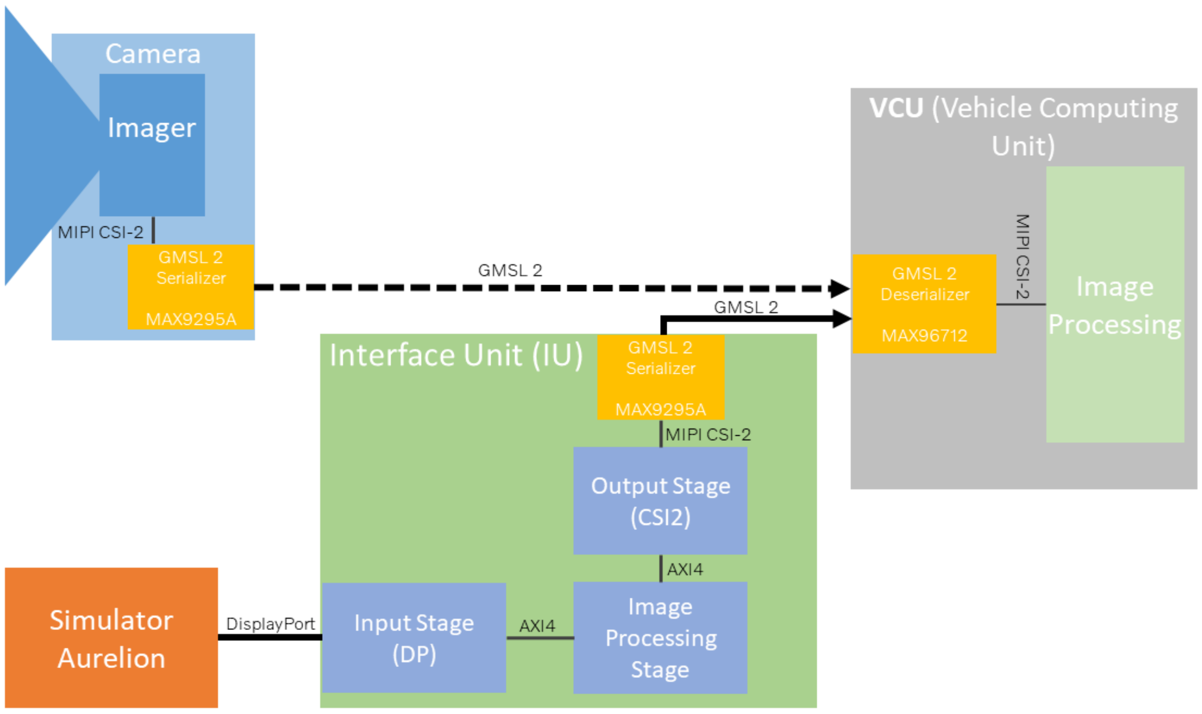


Figure 1.3: Block diagram of the image injection setup that details the important parts of the thesis

1.4 Division of the Thesis

The whole thesis project was concerned with the IU. It was conducted in cooperation by three students, the author of this thesis, Justus Hoffmann and Nidhi Chakrabhavi Basavaraju, and was therefore is divided into three separate parts. Referring to Figure 1.1, Justus' part mainly focuses on the input stage, Nidhi's part mainly focuses on the image processing stage, and my part mainly focuses on the output stage. Justus' report [6] and Nidhi's report [7] can both be found in the Reference section, since each of their reports will be referenced throughout this report when needed. Though, all of the work and theory presented from this point forward, was solely carried out by me.

2 Technical Background and Theory

2.1 Vehicle Communciation Architecture

Most, if not all, cars and vehicles that are currently being produced have a built in computer that handles most software logic in a car. This computer is a VCU. The VCU either directly controls features in a car that require computing logic or handles communication with other parts of the car that handles the computing logic, i.e the VCU can branch out to other computing units that do the computing more "locally". These units are usually called *Vehicle Interface Units* (VIUs). The VCU and VIUs are usually called the *Core System.* The VIUs are in turn, connected to multiple different ECUs. The ECUs are tailored to measure and control very specific part of the vehicle. For example one can monitor and/or control the engine, while another one can monitor and/or control braking. The ECUs are connected to what is called the *Mechatronic Rim.* Since the ECUs and VCU usually communicate via different network protocols/interfaces, the VIUs can be seen as a translator that handles the conversion. Usually the Mechantronic Rim uses *Controller Area Network* (CAN) or *Local Interconnect Network* (LIN) as communication networks and the Core System use Ethernet. However, when ECUs are directly connected to the VCU they need to operate on Ethernet to accommodate the VCU [8].

CAN and LIN are communication protocols/networks that are frequently used for electronic communication in vehicles. LIN can usually be found as the main communication protocols in older vehicles and larger trucks. Though, in modern cars it also provides a more cost effective and easier to use alternative, compared to the more reliable protocol CAN. This can be usefull in certain parts of the car that does not require high fault tolerance or precise timing. Rain sensors, window adjustments, and wipers are some example areas in the car that can use LIN [9].

2.2 FPGA, Microncontroller, and CPLD

Field Programmable Gate Arrays (FPGAs) and microcontrollers are two types of programmable integrated circuits (ICs) used in a wide range of applications. Both have their own unique strengths and weaknesses, and the choice between them depends on the specific requirements of the application. FPGAs are reconfigurable digital circuits that can be programmed to perform a wide range of tasks. They are created using a matrix of programmable logic blocks that can be connected together to form complex digital circuits. FPGAs are highly configurable and offer a high degree of flexibility, making them ideal for applications that require a high level of customization or fast prototyping. FPGAs also offer high speed and low latency, which makes them ideal for applications that require real-time processing, such as image and signal processing. On the other hand, microcontrollers are small, single-chip computers that are designed to perform specific tasks. They contain a CPU, memory, and input/output peripherals such as timers, UARTs and ADCs. Microcontrollers are ideal for applications that require low power consumption, low cost and low complexity. They are used in a wide range of embedded systems, such as consumer electronics, industrial control systems, and automotive applications [10].

Compared to an FPGA whose purpose is to be a configurable gate array with large logic resources that can be programmed by the user after manufacturing, a *Complex Programming Logic Device* (CPLD) is an IC that essentially assists implementation of digital systems or handles less computationally heavy operations. It can be seen as similar to an FPGA but with fewer logic resources and simpler interconnections. This makes the CPLD a more cost effective alternative to the FPGA. It is common for CPLDs used for digital circuit implementation (mostly in systems together with an FPGA) to have firmware pre-programmed, detailing the interconnections between the CPLD's pins. Though, the firmware can be re-programmed and so can the logic array within the CPLD [11][12][13].

2.3 MPSoC

A *System on a Chip* (SoC) refers to system, including a multitude of different peripheries and integrated circuits, built onto one single microchip. This differs from a traditional motherboard where the GPU, CPU, memories, peripherals, etc. all are separate components [14]. A *Multiprocessor System on a Chip* (MPSoC) is a *System on a Chip* with multiple microprocessors instead of one, which allow for multi-processing capabilities [15].

2.4 AMD Vivado Design Suite

Most synthesis tools on the market aimed at programming FPGAs expects the design provided, as input, to be on *Register Transfer Level* (RTL) form [16]. Vivado is a software/IDE used for RTL code development, anaysis, synthesis, and implementation. It operates on an RTL-to-Bitstream design flow, which means that RTL designs can be developed, synthesised, implemented, and generated into a bitstream. This bitstream can later be used to program the FPGA [17].

Since Vivado is a software that aims at programming of FPGAs and MPSoCs, the main languages available are VHDL and Verilog. While these can be used to write custom code, most of the designs developed in Vivado are based on *Intelectual Property* (IP) cores.

2.5 IP Cores and Subsystems

An IP core can be seen as a pre-programmed block of code that, thanks to Vivado's IP integrator, can easily be configured either through an interactive GUI or programmatically through TCL programming interface. By introducing these pre-programmed blocks of code that are proven to work with Xilinx devices and are easily adjustable for the user, it increases the development simplicity and re-usability while decreasing development time. Vivado also offers numerous IP Subsystems that provide a single solution with multiple integrated IP cores, interconnected using AMBA AXI4 interconnect protocol. These IP cores and subsystems can, in turn, easily be connected together in a Vivado *block diagram*, that is used to build a full RTL design [17][18].

2.6 Communication Protocols and Connections

2.6.1 AMBA AXI4

AMBA *Advanced eXtensible Interface 4* (AXI4) is an interface specification developed by ARM. The AMD Vivado Design Suite extends development capabilities with the AXI4 interface, providing users of Xilinx devices a standardized and easy to use protocol for IP core integration [19]. There are three main AXI4 interfaces that each have their own area of use. The first one is AXI4, or Full AXI4. This interface is more commonly used for high-performance memory mapped requirements. The second interface is the AXI4-Lite. This protocol aims more towards simpler memory mapped communication requiring low-throughput, and can usually be found handling control and status registers. The final interface is the AXI4-Stream, that targets high speed streaming data. The AXI4 protocol as a whole, is a master/slave-based architecture where the master initiates communication and the slave responds. Additionally, the AXI4 is a channel-based interface. This means that the data transferred between the master and the slave uses a set of five independent channels for reading and writing (refer to Figure 2.1) [20].

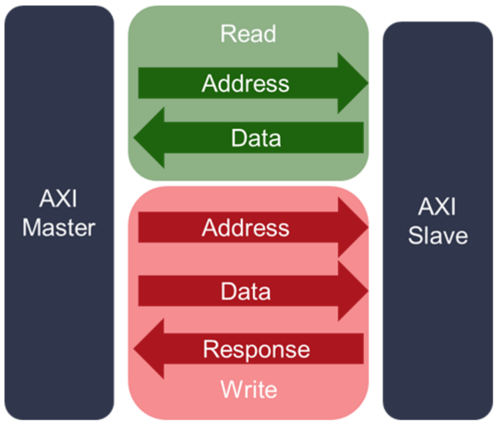


Figure 2.1: AXI interface with five read and write channels [20]

Data being transmitted on one of the channels is called a transfer. A transfer only occurs upon the rising edge of the clock when both signals READY and VALID are high. As can be observed in Figure 2.2 a transfer occurs at time instance T3 [20]. Note that ACLK here is the AXI reference clock.

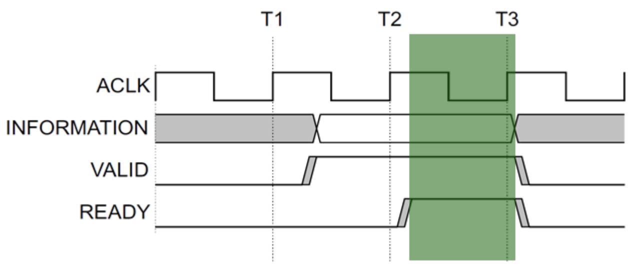


Figure 2.2: AXI4 information transfer with time on the x-axis [20]

The AXI4 Read/Write transaction works as follows. Firstly, for a simplified depiction of the read transaction, observing Figure 2.3 and 2.1, the master sends information regarding the address and control signals via the Address Read Channel. Then from the slave the data for the address is transmitted to the master via the Read Data Channel. This read transaction requires multiple transfers [20].

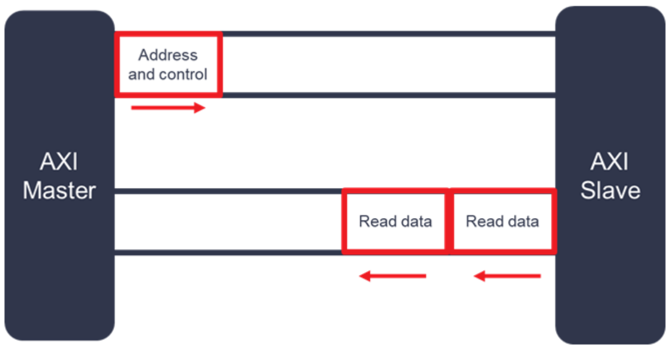


Figure 2.3: AXI4 Read Transaction [20]

Secondly, for a simplified depiction of the write transaction, observing Figure 2.4 and 2.1, the master initially sends information regarding address and control signals via the Address Write Channel. The data correlated to this address is then transmitted from the master to the slave via the Write Data Channel. Lastly, the response from the slave to the master is sent via the Write Response Channel detailing whether the transfer is successful or not. This write transaction also requires multiple transfers [20].

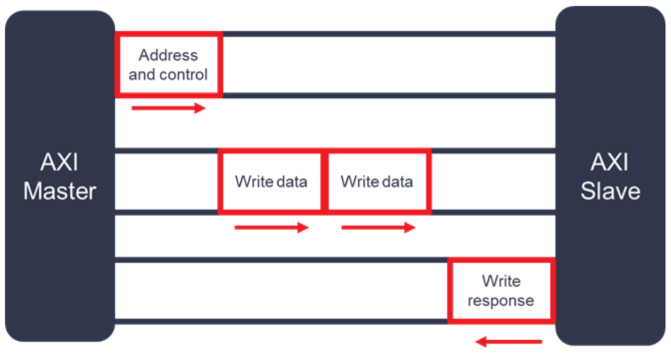


Figure 2.4: AXI4 Write Transaction [20]

2.6.2 Serial Communication and SerDes

In serial communication, the clock and the data are combined into one signal instead of sending them in separate signals where the clock edge samples the data line, as is the case in parallel busses. This is done by encoding schemes. The encoder (transmitter) and the decoder (receiver) use the same scheme, and the receiver extracts the clock from the data. Hence, the clock is still needed on receiver side to sample the data line, but the receiver can get its own clock. Most common encoding is the 8B/10B, where 8bit of data is converted into 10 bits of data effectively losing 25% bandwidth, but it does guarantee DC balance of the line and transitions for Clock/Data Recovery (CDR). Where DC balance is the avoidance of DC bias, which can occur for example if multiple logical 1's are sent in a row causing a charge/bias to build up in the line [21]. For transmitting and receiving the serial communication there needs to be a serializer (encoder) on the transmitting side and a deserializer (decoder) on the receiving side. In serial communication it is important to match the Serializer and the Deserializer so that they operate on the matching encoding/decoding schemes.

2.6.3 MIPI CSI-2

CSI-2 or Camera Serial Interface-2 is an image sensor interface and is commonly used for cameras in phones, tablets and automotive applications. CSI-2 essentially provides a simple sensor interface that allows for high bit rate streaming. The protocol is usually implemented with the physical layer MIPI D-PHY, and the standardized CSI-2 cable has the following channels:

- 2 x GPIO for Camera Control (I^2C)
- 1 x Clock lane
- 4 x Image data lanes (Scalable)
- 2 x Free GPIO (e.g "for external trigger signals or peripheral control")
- 1 x External power

Hence, there is a total of 10 lanes running along the CSI-2 cable. The data lanes of the CSI-2 are scalable, which means that a choice can be made to use any number of data lanes. Since each data lane can transfer data up to 1.5Gbps and there are a total of 4 data lanes, the bandwidth can be adjusted between 0.2Gbps to 6Gbps [22]. An example of the 4 lane data stream for CSI-2 is shown in Figure 2.5.

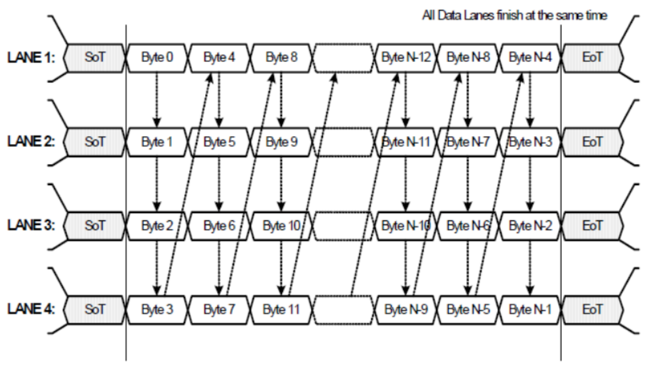


Figure 2.5: CSI-2 4-lane data stream [23]

Cameras for AD and ADAS frequently use MIPI CSI-2 as the protocol for communicating the image data from the imager of the camera to other components of the system. Since this is the case, the image processing stack of the VCU only accepts image data in strictly the CSI-2 format. Thus, to inject images into such a VCU the data needs to be on the CSI-2 format.

2.6.4 GMSL2

Gigabit Multimedia Serial Link (GMSL) are high speed communication ICs that use serializing and deserializing (SerDes) technology to provide multistream support over a single cable. SerDes effectively reduces the number of cables/IO pins needed, reduces the risk for clock skew during high clock speeds, generally provides much faster bit rates, and allows high speed data transmission over longer distances compared to wide parallel busses. The GMSL is tailored to handle multiple data streams, which makes it suitable for splitting image data streams. Also, it only requires one link to convert other protocols such as DisplayPort. The difference between GMSL and GMSL2 is that GMSL2 supports 4k video support while GMSL only supports 1080p. It also supports bidirectional diagnostic, which can be useful during testing. GMSL2 is a suitable solution for Automotive Infotainment and ADAS since it achieves the requirements for data integrity, complex interconnects and high bandwidth [24].

2.7 Image Theory

2.7.1 Camera Basics

Images from a camera can be captured in multiple different way. However, digital cameras are usually based on photosensetive CMOS layer that is divided into a grid. The image that the camera captures depend on the so called Color Filter Array (CFA) in the camera. This is an arrangement of color filters placed over the sensor grid of the camera to capture the RGB color information. Since most cameras do not have the capability of capturing more than one color per square/position in the grid, a color filter array has to be chosen for that particular camera. Each of these position in the grid essentially represents a pixel of the image [25]. A clear representation of this is ullustrated in Figure 2.6.

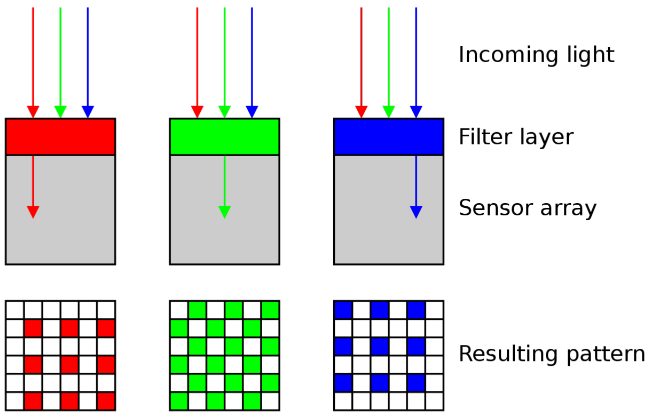


Figure 2.6: Graphical representation of a CFA [26]

2.7.2 Bayer Filter

One of the more common CFAs is the Bayer Filter. The Bayer Filter color array is structured in a way that half of the pixels represent a green value, one fourth of the pixels represent a red value, and one fourth of the pixels represent a blue value. This is usually called RGGB. The pattern is structured in such a way that each line either alternates between green and red or green and blue. For example, the first pixel in the first row starts with a blue pixel, then the next pixel in the same row is a green pixel, then the next is a blue pixel, etc, until the end of the line. Then in the second line, the first pixel is green and the rest of the pixels in this line alternate between red and green instead. Essentially producing a similar color format as the one in Figure 2.7. The starting pixel is not of importance, but rather the pattern that the pixel color is repeated with, each line alternating between either green and blue or green and red [25][27].

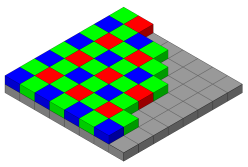


Figure 2.7: Example of a Bayer Pattern CFA [26]

2.7.3 RAW

Unprocessed image data directly captured from the camera, such as image data in RGGB format, is commonly called a RAW image. Depending on the amount of bits used to represent each pixel a number is included in the name. Thus, RAW8 has 8-bit pixel representation, RAW10 has 10-bit pixels, RAW12 has 12-bit pixels, etc [28].

2.7.4 RGB

When using a CFA such as the Bayer Filter a lot of information can be lost, since each pixel only represents one color. This is where de-mosaicing is useful. De-mosaicing is the process of essentially interpolating the pixel values around a pixel that lack the color information. For example, a green pixel in a RGGB pattern have neighboring blue and red pixels. An algorithm can then be implemented that interpolates the values of the sorrounding red and blue values so that the green pixel gets an estimated value of what the red an blue values would be at that specific place in the picture. This is essentially how a common RGB image is produced [25]. RGB usually has a 8-bit representation of each of the colors (red, green, blue) for each of the pixels in the image. This is often referred to as RGB888 and is typically the image data format that is transmitted over for example DisplayPort or HDMI.

3 Method

3.1 Hardware

Usually when it comes to programming hardware, a relatively easily programmed computing unit is needed. The two more common types of programmable computing units are Microcontrollers and FPGAs. Projects requiring high computational complexity, parallel computing and/or greater customizability usually benefits from using an FPGA. This is because an FPGA can run processes in parallel, enabling higher data rates and computational capacity. FPGAs also have higher customizability than a standard Microcontroller, since changes can be made inside the actual hardware [10]. Since HIL setups often requires high bit rates and complex operations with specific requirements, an FPGA-based board is more suitable for this project. Though, a Microcontroller can probably still be used to accomplish image injection, though preferably not in high demand environments.

3.1.1 Zynq™ UltraScale+ MPSoC SOM

The specific MPSoC used in this project is the Zynq™ Ultrascale+™ MPSoC [29], developed by AMD Xilinx. This MPSoC is based on the Zynq™ Ultrascale+™ FPGA. The MPSoC is equipped with a *Processing System* (PS) bridged with a *Programmable Logic* (PL), allowing the user to run either sequential logic on the PS or parallel logic on the PL. So, the PS can be programmed with C/C++ while the PL can be programmed with VHDL/Verilog. A more detailed view of the Zynq UltraScale+ MPSoC's composition is presented in Figure 3.1. A picture of the Zynq UltraScale+ MPSoC can be seen in Figure 3.2.

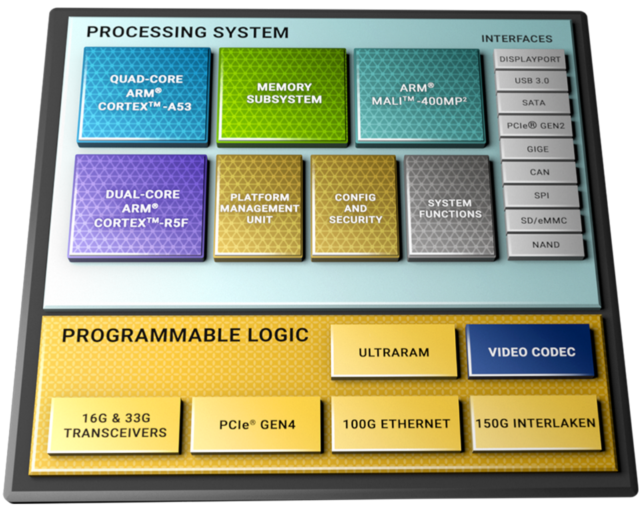


Figure 3.1: Zynq UltraScale+ MPSoC composition [29]

Figure 3.2: Zynq UltraScale+ MPSoC [30]

The MPSoCs are usually not sold piece-wise, but rather implemented onto a board/module that is designed for a specific application area. This is a so called *System-on-Module* (SOM) [15]. The specific

SOM used in this degree project is the TEB0911 UltraRack+ MPSoC Board [31], produced and manufactured by Trenz Electronic. This specific board was chosen because it was readily available and proven to be a suitable IU solution. The TEB0911 module can be seen in Figure 3.3. There is a multitude of important information regarding the TEB0911 that will be conveyed in other parts of the Method, where the information is more relevant.

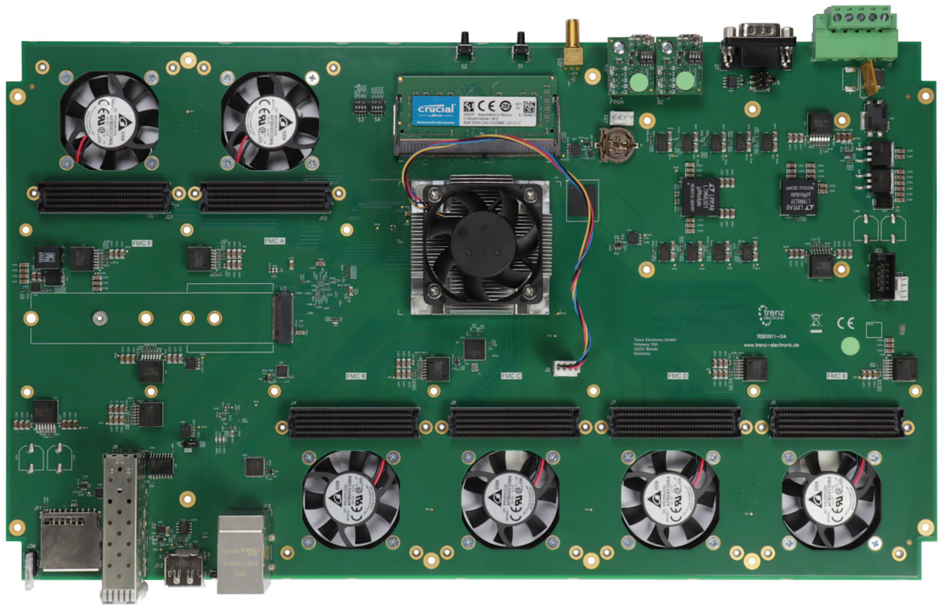


Figure 3.3: TEB0911 UltraRack+ MPSoC SOM [31]

For brevity, the "Zynq UltraScale+ MPSoC" will at times be refered to as "Ultrascale+ MPSoC" or just "MPSoC". The Zynq UltraScale+ MPSoC SOM "TEB0911 UltraRack+ MPSoC" will be refered to as "TEB0911". The "Zynq UltraScale+ MPSoC FPGA" will at times be refered to as "UltraScale+ FPGA", "MPSoC FPGA", or "TEB0911 FPGA".

3.1.2 FMC connectors

The TEB0911 module is equipped with six *FPGA Mezzanine Card* (FMC) connectors, which can be seen in Figure 3.3. The FMC connector is a standardized FPGA Input/Output (I/O) connection socket that makes I/O connections to FPGA SoC boards easier [32]. A more specific representation of the main components of the TEB0911 module is presented in Figure 3.4. The FMC connectors range from A-F, and each have a different set of capabilities. The FMC modules are then connected to one of these FMC sockets depending on the specific requirements of the FMC module. A list can be seen below, detailing which FMC corresponds to which number in Figure 3.4 [33].

4. FMC B
6. FMC C
8. FMC D
10. FMC E
27. FMC A
29. FMC F

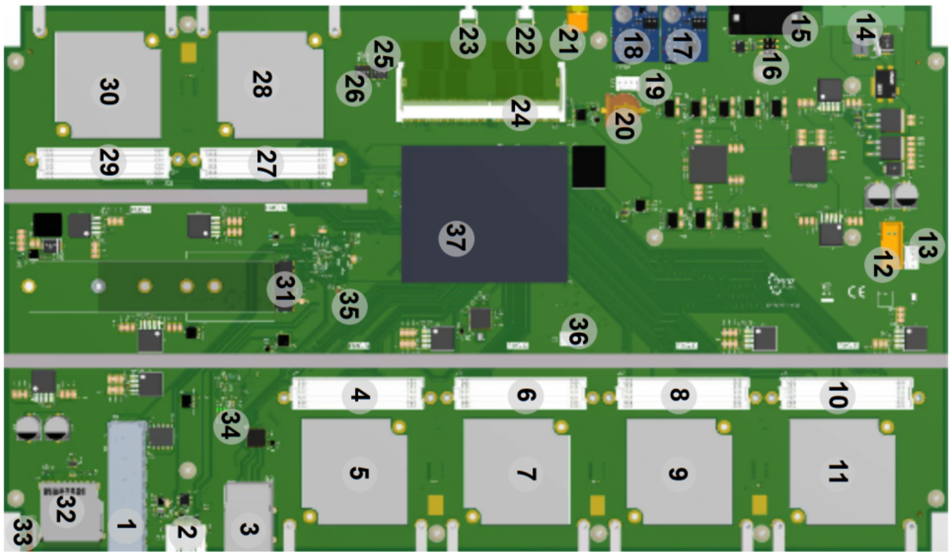


Figure 3.4: TEB0911 main components [33]

FMC connectors can be divided into two groups; *High-pin count* (HPC) and *Low-pin count* (LPC). The HPC and LPC connectors have the same form factor, but the HPC has more usable pins than the LPC. An HPC FMC connector has 10 rows (A, B, C, D, E, F, G, H, J, K) with 40 pins on each row (A1-A40, B1-B40, etc.), while a LPC FMC connector only has 4 rows (C, D, G, H) with 40 pins on each row [34]. The socket/connector can be seen in Figure 3.5.

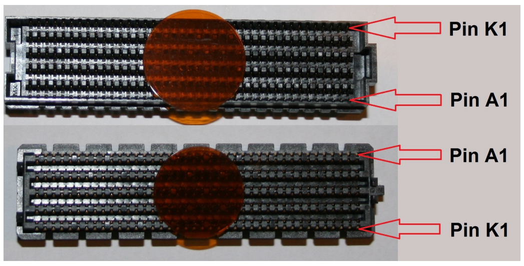
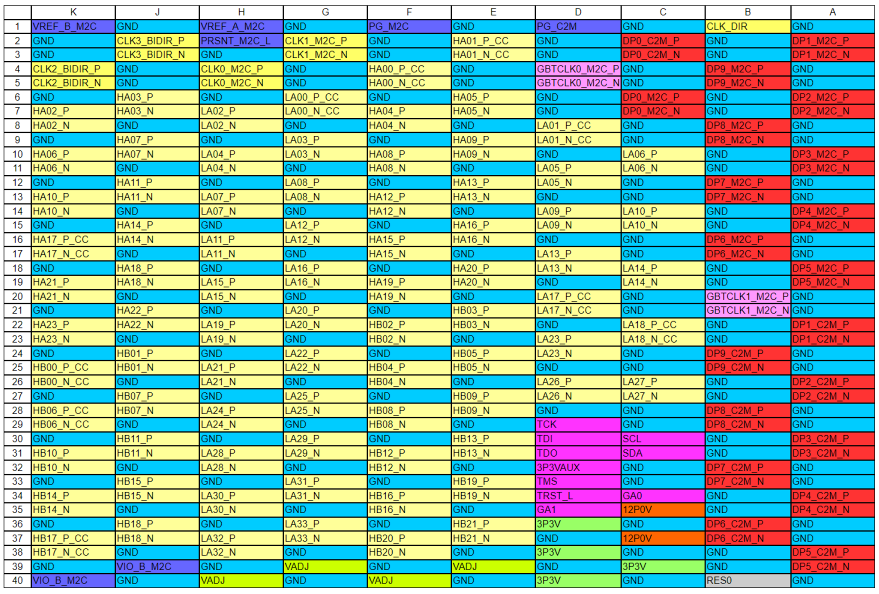


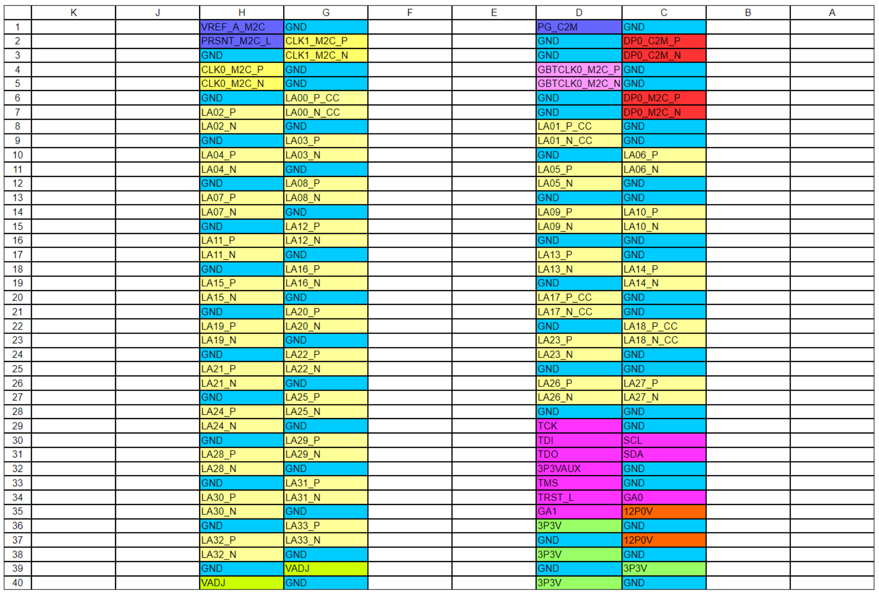
Figure 3.5: FMC HPC socket [34]

The pinouts for the HPC FMC can be observed in Figure 3.6. Each pin is labeled and colored depending on its protocol and what signals it is capable of transmitting/receiving. The pinouts for LPC FMC can be seen in Figure 3.7, as a comparison. A summary of the total number of pin types for the HPC FMC can be seen in Figure 3.8. Pin descriptions for some of the more important pins is presented in Figure 3.9. Note the `LA_XX` pins, which will be seen in later sections of the report due to its use in testing of CSI2 data and clock transmitting. It is important to know each of the pin on the FMC connector to be able to map the signals to the appropriate pins on the FPGA.



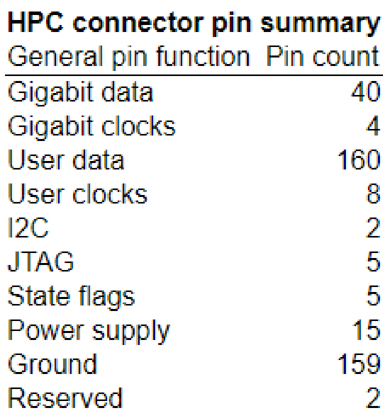
	K	J	H	G	F	E	D	C	B	A
1	VREF_B_M2C	GND	VREF_A_M2C	GND	PG_M2C	GND	PG_C2M	GND	CLK_DIR	GND
2	GND	CLK3_BIDIR_P	PRSNT_M2C_L	CLK1_M2C_P	GND	HA01_P_CC	GND	DP0_C2M_P	GND	DP1_M2C_P
3	GND	CLK3_BIDIR_N	GND	CLK1_M2C_N	GND	HA01_N_CC	GND	DP0_C2M_N	GND	DP1_M2C_N
4	CLK2_BIDIR_P	GND	CLK0_M2C_P	GND	HA00_P_CC	GND	GBTCLK0_M2C_P	GND	DP9_M2C_P	GND
5	CLK2_BIDIR_N	GND	CLK0_M2C_N	GND	HA00_N_CC	GND	GBTCLK0_M2C_N	GND	DP9_M2C_N	GND
6	GND	HA03_P	GND	LA00_P_CC	GND	HA05_P	GND	DP0_M2C_P	GND	DP2_M2C_P
7	HA02_P	HA03_N	LA02_P	LA00_N_CC	HA04_P	HA05_N	GND	DP0_M2C_N	GND	DP2_M2C_N
8	HA02_N	GND	LA02_N	GND	HA04_N	GND	LA01_P_CC	GND	DP8_M2C_P	GND
9	GND	HA07_P	GND	LA03_P	GND	HA09_P	LA01_N_CC	GND	DP8_M2C_N	GND
10	HA06_P	HA07_N	LA04_P	LA03_N	HA08_P	HA09_N	GND	LA06_P	GND	DP3_M2C_P
11	HA06_N	GND	LA04_N	GND	HA08_N	GND	LA05_P	LA06_N	GND	DP3_M2C_N
12	GND	HA11_P	GND	LA08_P	GND	HA13_P	LA05_N	GND	DP7_M2C_P	GND
13	HA10_P	HA11_N	LA07_P	LA08_N	HA12_P	HA13_N	GND	GND	DP7_M2C_N	GND
14	HA10_N	GND	LA07_N	GND	HA12_N	GND	LA09_P	LA10_P	GND	DP4_M2C_P
15	GND	HA14_P	GND	LA12_P	GND	HA16_P	LA09_N	LA10_N	GND	DP4_M2C_N
16	HA17_P_CC	HA14_N	LA11_P	LA12_N	HA15_P	HA16_N	GND	GND	DP6_M2C_P	GND
17	HA17_N_CC	GND	LA11_N	GND	HA15_N	GND	LA13_P	GND	DP6_M2C_N	GND
18	GND	HA18_P	GND	LA16_P	GND	HA20_P	LA13_N	LA14_P	GND	DP5_M2C_P
19	HA21_P	HA18_N	LA15_P	LA16_N	HA19_P	HA20_N	GND	LA14_N	GND	DP5_M2C_N
20	HA21_N	GND	LA15_N	GND	HA19_N	GND	LA17_P_CC	GND	GBTCLK1_M2C_P	GND
21	GND	HA22_P	GND	LA20_P	GND	HB03_P	LA17_N_CC	GND	GBTCLK1_M2C_N	GND
22	HA23_P	HA22_N	LA19_P	LA20_N	HB02_P	HB03_N	GND	LA18_P_CC	GND	DP1_C2M_P
23	HA23_N	GND	LA19_N	GND	HB02_N	GND	LA23_P	LA18_N_CC	GND	DP1_C2M_N
24	GND	HB01_P	GND	LA22_P	GND	HB05_P	LA23_N	GND	DP9_C2M_P	GND
25	HB00_P_CC	HB01_N	LA21_P	LA22_N	HB04_P	HB05_N	GND	GND	DP9_C2M_N	GND
26	HB00_N_CC	GND	LA21_N	GND	HB04_N	GND	LA26_P	LA27_P	GND	DP2_C2M_P
27	GND	HB07_P	GND	LA25_P	GND	HB09_P	LA26_N	LA27_N	GND	DP2_C2M_N
28	HB06_P_CC	HB07_N	LA24_P	LA25_N	HB08_P	HB09_N	GND	GND	DP8_C2M_P	GND
29	HB06_N_CC	GND	LA24_N	GND	HB08_N	GND	TCK	GND	DP8_C2M_N	GND
30	GND	HB11_P	GND	LA29_P	GND	HB13_P	TDI	SCL	GND	DP3_C2M_P
31	HB10_P	HB11_N	LA28_P	LA29_N	HB12_P	HB13_N	TDO	SDA	GND	DP3_C2M_N
32	HB10_N	GND	LA28_N	GND	HB12_N	GND	3P3VAUX	GND	DP7_C2M_P	GND
33	GND	HB15_P	GND	LA31_P	GND	HB19_P	TMS	GND	DP7_C2M_N	GND
34	HB14_P	HB15_N	LA30_P	LA31_N	HB16_P	HB19_N	TRST_L	GA0	GND	DP4_C2M_P
35	HB14_N	GND	LA30_N	GND	HB16_N	GND	GA1	12P0V	GND	DP4_C2M_N
36	GND	HB18_P	GND	LA33_P	GND	HB21_P	3P3V	GND	DP6_C2M_P	GND
37	HB17_P_CC	HB18_N	LA32_P	LA33_N	HB20_P	HB21_N	GND	12P0V	DP6_C2M_N	GND
38	HB17_N_CC	GND	LA32_N	GND	HB20_N	GND	3P3V	GND	GND	DP5_C2M_P
39	GND	VIO_B_M2C	GND	VADJ	GND	VADJ	GND	3P3V	GND	DP5_C2M_N
40	VIO_B_M2C	GND	VADJ	GND	VADJ	GND	3P3V	GND	RES0	GND

Figure 3.6: FMC HPC pinout [34]



	K	J	H	G	F	E	D	C	B	A
1			VREF_A_M2C	GND			PG_C2M	GND		
2			PRSNT_M2C_L	CLK1_M2C_P			GND	DP0_C2M_P		
3			GND	CLK1_M2C_N			GND	DP0_C2M_N		
4			CLK0_M2C_P	GND			GBTCLK0_M2C_P	GND		
5			CLK0_M2C_N	GND			GBTCLK0_M2C_N	GND		
6			GND	LA00_P_CC			GND	DP0_M2C_P		
7			LA02_P	LA00_N_CC			GND	DP0_M2C_N		
8			LA02_N	GND			LA01_P_CC	GND		
9			GND	LA03_P			LA01_N_CC	GND		
10			LA04_P	LA03_N			GND	LA06_P		
11			LA04_N	GND			LA05_P	LA06_N		
12			GND	LA08_P			LA05_N	GND		
13			LA07_P	LA08_N			GND	GND		
14			LA07_N	GND			LA09_P	LA10_P		
15			GND	LA12_P			LA09_N	LA10_N		
16			LA11_P	LA12_N			GND	GND		
17			LA11_N	GND			LA13_P	GND		
18			GND	LA16_P			LA13_N	LA14_P		
19			LA15_P	LA16_N			GND	LA14_N		
20			LA15_N	GND			LA17_P_CC	GND		
21			GND	LA20_P			LA17_N_CC	GND		
22			LA19_P	LA20_N			GND	LA18_P_CC		
23			LA19_N	GND			LA23_P	LA18_N_CC		
24			GND	LA22_P			LA23_N	GND		
25			LA21_P	LA22_N			GND	GND		
26			LA21_N	GND			LA26_P	LA27_P		
27			GND	LA25_P			LA26_N	LA27_N		
28			LA24_P	LA25_N			GND	GND		
29			LA24_N	GND			TCK	GND		
30			GND	LA29_P			TDI	SCL		
31			LA28_P	LA29_N			TDO	SDA		
32			LA28_N	GND			3P3VAUX	GND		
33			GND	LA31_P			TMS	GND		
34			LA30_P	LA31_N			TRST_L	GA0		
35			LA30_N	GND			GA1	12P0V		
36			GND	LA33_P			3P3V	GND		
37			LA32_P	LA33_N			GND	12P0V		
38			LA32_N	GND			3P3V	GND		
39			GND	VADJ			GND	3P3V		
40			VADJ	GND			3P3V	GND		

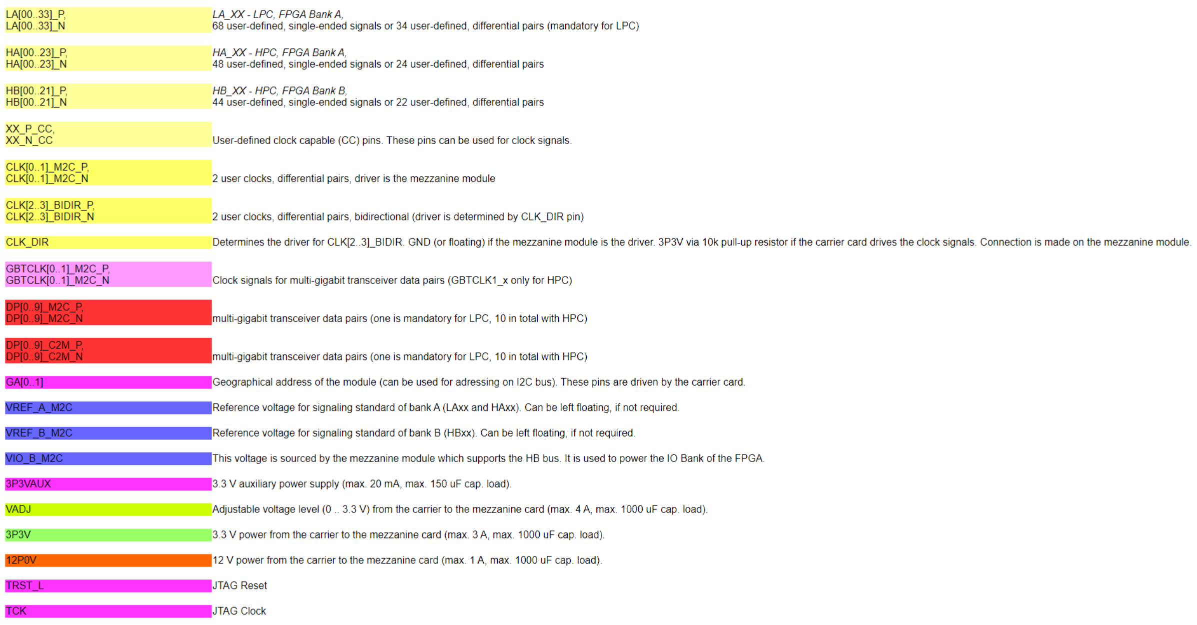
Figure 3.7: FMC LPC pinout [34]



HPC connector pin summary

General pin function	Pin count
Gigabit data	40
Gigabit clocks	4
User data	160
User clocks	8
I2C	2
JTAG	5
State flags	5
Power supply	15
Ground	159
Reserved	2

Figure 3.8: HPC connector pin summary [34]



LA[00..33]_P, LA[00..33]_N	*LA_XX - LPC, FPGA Bank A,* 68 user-defined, single-ended signals or 34 user-defined, differential pairs (mandatory for LPC)
HA[00..23]_P, HA[00..23]_N	*HA_XX - HPC, FPGA Bank A,* 48 user-defined, single-ended signals or 24 user-defined, differential pairs
HB[00..21]_P, HB[00..21]_N	*HB_XX - HPC, FPGA Bank B,* 44 user-defined, single-ended signals or 22 user-defined, differential pairs
XX_P_CC, XX_N_CC	User-defined clock capable (CC) pins. These pins can be used for clock signals.
CLK[0..1]_M2C_P, CLK[0..1]_M2C_N	2 user clocks, differential pairs, driver is the mezzanine module
CLK[2..3]_BIDIR_P, CLK[2..3]_BIDIR_N	2 user clocks, differential pairs, bidirectional (driver is determined by CLK_DIR pin)
CLK_DIR	Determines the driver for CLK[2..3]_BIDIR. GND (or floating) if the mezzanine module is the driver. 3P3V via 10k pull-up resistor if the carrier card drives the clock signals. Connection is made on the mezzanine module.
GBTCLK[0..1]_M2C_P, GBTCLK[0..1]_M2C_N	Clock signals for multi-gigabit transceiver data pairs (GBTCLK1_x only for HPC)
DP[0..9]_M2C_P, DP[0..9]_M2C_N	multi-gigabit transceiver data pairs (one is mandatory for LPC, 10 in total with HPC)
DP[0..9]_C2M_P, DP[0..9]_C2M_N	multi-gigabit transceiver data pairs (one is mandatory for LPC, 10 in total with HPC)
GA[0..1]	Geographical address of the module (can be used for adressing on I2C bus). These pins are driven by the carrier card.
VREF_A_M2C	Reference voltage for signaling standard of bank A (LAxx and HAxx). Can be left floating, if not required.
VREF_B_M2C	Reference voltage for signaling standard of bank B (HBxx). Can be left floating, if not required.
VIO_B_M2C	This voltage is sourced by the mezzanine module which supports the HB bus. It is used to power the IO Bank of the FPGA.
3P3VAUX	3.3 V auxiliary power supply (max. 20 mA, max. 150 uF cap. load).
VADJ	Adjustable voltage level (0 .. 3.3 V) from the carrier to the mezzanine card (max. 4 A, max. 1000 uF cap. load).
3P3V	3.3 V power from the carrier to the mezzanine card (max. 3 A, max. 1000 uF cap. load).
12P0V	12 V power from the carrier to the mezzanine card (max. 1 A, max. 1000 uF cap. load).
TRST_L	JTAG Reset
TCK	JTAG Clock

Figure 3.9: FMC pinout descriptions [34]

3.1.3 Input Stage: DP FMC card

One of the main parts of the degree project is to investigate how to handle the input stage. This part is the main topic of Justus' thesis report [6]. Since the sensor simulation PC outputs image data via DisplayPort, it advantageous to adapt the IU to handle the DisplayPort protocol as an input. To achieve proper input speeds and to allow multiple DisplayPort connections to the IU, an FMC module needs to be connected to the TEB0911 module. The main responability of this module is to convert the DisplayPort format to an AXI data stream that can be transferred to the UltraScale+ MPSoC FPGA.

The DP FMC module used in this project is the TEF0007 [35], which is developed and sold by Trenz Electronics. The card offers DisplayPort 1.2 support, with one input and one output port. It is based on a Xilinx Kintex-7 FPGA [36], so the module can be programmed and perform the needed operations locally, instead of on the MPSoC FPGA. The card can be seen in Figure 3.10.

Figure 3.10: TEF0007 DisplayPort FMC module [37]

3.1.4 Output Stage: GMSL2 FMC card

In the same way as the input stage, how to handle the output stage needs to be investigated. This part of the degree project was the main topic of this specific thesis project and report.

As mentioned in Section 1.2 communication between the IU and the VCU is transmitted via fiber-optic cable in GMSL2 format. Since GMLS2 is a serial communication protocol, the image data transmitted from the IU needs to be serialized. This is accomplished with the assistance of serializers and deserializers. As mentioned in the Section 2.6.2, serializer/deserializer (SerDes) pairs needs to match in order to communicate properly. The deserializer on the receiving side of the GMLS2 cable, inside the VCU, is a MAX96712. By default the serializer on the transmitting side, inside the IU, needs to be a MAX9295A to match the deserializer. Refer to Figure 1.3 for a clear illustration of this SerDes pair. Since the serializers on the transmitting side take CSI2 as input, the main responsibility of this module is to convert the AXI4 data stream from the UltraScale+ MPSoC FPGA to CSI2 format that can be accepted by the serializers and converted into GMSL2 output.

The specific FMC module that most of the work is based on is the TEF0010 [38], which is developed and sold by Trenz Electronics. The card offers four GMSL2 ouputs, each with its own dedicated serializer. It is based on a Xilinx Artix-7 FPGA [39], so the module can be programmed and perform the needed operations locally, instead of on the UltraScale+ MPSoC FPGA. The card can be seen in Figure 3.11.

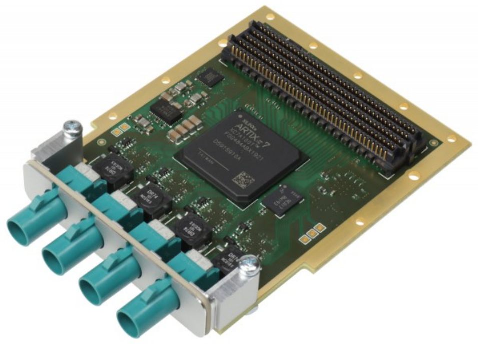


Figure 3.11: TEF0010 GMSL2 FMC module [40]

3.1.5 Bridging Input and Output

The bridging of the input and output stage on the hardware level is accomplished by connecting each of the FMC modules to the FMC connectors/sockets on the UltraScale+ MPSoC SOM TEB0911. Which connector to use, out of the six available on the TEB0911, depends on the specific requirements of the FMC module. The TEF0007 was mounted on the FMC A connector and the TEF0010 was mounted on the FMC E connector. Since the mounting of the FMC modules onto the TEB0911 were handled by a different company working closely with the provider of the cards, no definite motivation for this decision can be given.

However, bridging the input and output stages on a software level is more difficult. Since the sensor simulation PC provides image data in RGB color format via DisplayPort and the VCU only accepts image data in the RAW12 format, another major part of the degree project is handling the conversion between the two image formats. This part will be the main topic of Nidhi's thesis report [7].

The conversion between RAW-RGB is a frequent occurrence in many technologies. One example of this is digital cameras, which usually take the picture in RAW format and convert it into RGB to display the picture on the built in display [41]. However, the conversion from RGB-RAW is significantly more difficult. The algorithm that handles this image processing/conversion will be running on the UltraScale+ MPSoC FPGA.

3.1.6 SD card vs JTAG

There are usually multiple ways to configure programmable computational units such as Microcontrollers and FPGAs. One of the more popular alternatives is JTAG. This approach usually requires connectivity via some sort of cable, e.g USB-cable, to a PC. However, there are other options that can circumvent the need for cable connectivity.

The TEB0911 provides essentially two different option for programming of the MPSoC and FPGAs in the system: SD card and JTAG. The SD card can be used when automation is of importance since no external PC needs to be connected to the module. Instead, a bitstream is loaded onto the SD card that can be fetched directly by the MPSoC/FPGAs on start-up, if the SD card is plugged in to its slot in the TEB0911. However, this limits the possibility for live diagnostics and testing. This is where the JTAG connectivity is useful. With JTAG, the MPSoC and FPGAs can be programmed directly from an external PC with USB connectivity. Additionally, information can simultaneously be read from the module with serial monitoring, which provides an avenue for debugging and hardware testing. This makes programming via JTAG a much more suitable approach during development and testing.

Furthermore, the TEB0911 has two separate ports for JTAG connectivity; one for programming the MPSoC (XMOD JTAG Header J24) and one for programming the FPGAs on the FMC modules connected to the TEB0911 (XMOD JTAG Header J35). More specifically, the JTAG Header J35 programs the CPLD on the TEB0911. The CPLD in turn handles the programming of the FPGAs on the FMC modules connected to the TEB0911.

Whether the MPSoC on the TEB0911 and the FPGAs on the FMC modules are programmed via JTAG or SD card is decided by the configuration of the physical 4-bit *Dual in-line Package* (DIP) switch `S3` on the TEB0911 board. This switch also specifies whether the FPGAs of the FMC modules should be connected to the "JTAG-chain" of the CPLD, so that they can be programmed through the CPLD and JTAG Header J35. Each bit in the 4-bit `S3` switch has its own physical switch, resulting in `S3-1`, `S3-2`, `S3-3`, and `S3-4` being the names of the four individually configurable switches [33]. Refer to Figure 3.12 for an illustration of the S3 switch in the "layout schematic" of the TEB0911. Note that the top of the figure is one of the sides of the TEB0911 board, and `S2` for example is an external connector. Also note that the bit-switch `S3-1` is to the left and bit-switch `S3-4` is to the right, on the `S3` switch. For a better illustration of where the switch is situated on the TEB0911, refer to Figure 3.4 and position 26.

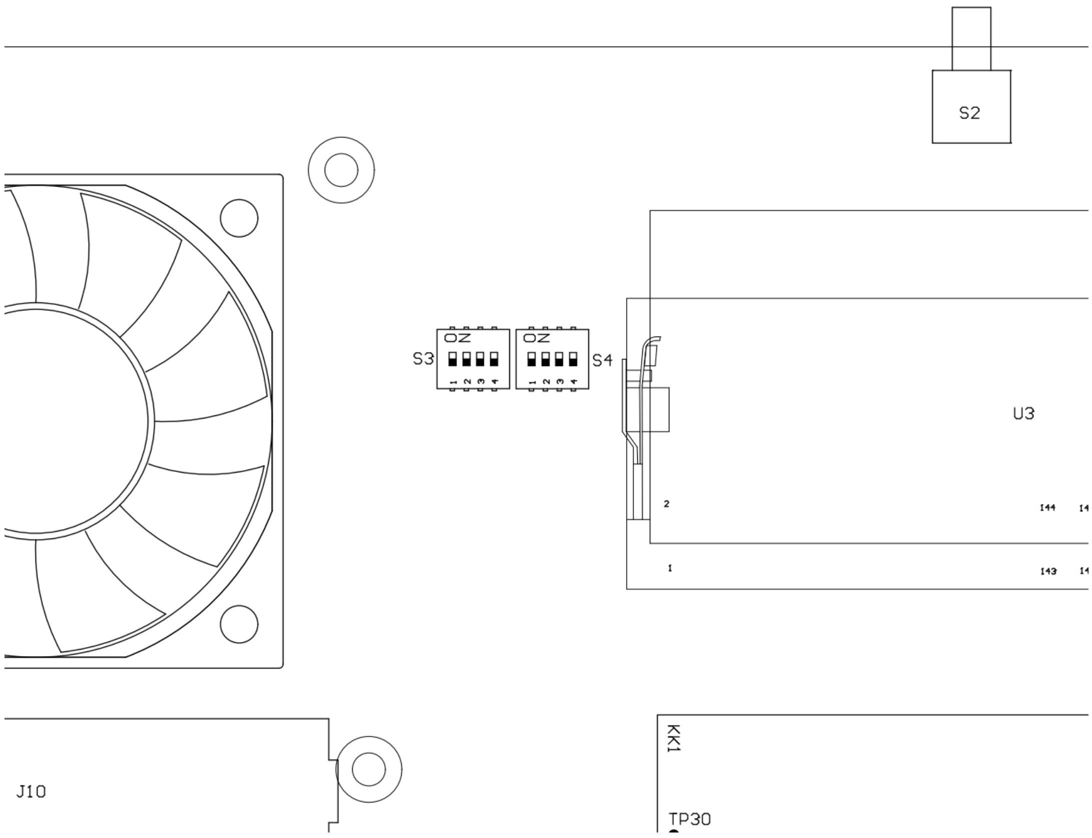


Figure 3.12: Layout schematic of the TEB0911 specifically showing the 4-bit `S3` switch [42]

The exact configuration used for the switches in thisproject to be able to program both the TEB0911 FPGA and the FPGAs on the FMC modules via JTAG is the following:

S3-1: '0'

S3-2: '0'

S3-3: '1'

S3-4: '1'

Comparing Figure 3.12 and the switch configuration above, note that setting the switch to '0' is down in the figure (towards the center of the TEB0911) and a '1' is up in the figure (away from the center of the TEB0911). I.e a '1' is toward the "ON" writing on the switch.

3.2 Software

The main problem of all three parts of the degree project will be how to program the software that will run on the hardware. This section will give an overview of the software and how the software is used to achieve the final results of the thesis.

3.2.1 Vivado

Since most of the hardware used in the project is produced by Xilinx, a software developed by Xilinx called Vivado was used to program the MPSoC and the FPGAs. More specifically the 2019.2 [43] version of Vivado. This particular version was chosen because it was the latest version of Vivado that undoubtedly supported the TEB0911, since there were reference designs available for for the TEB0911 in the 2019.2 version [44]. An attempt was made to use the 2022.2 version, but complications arose where it was not possible to include the Zynq UltraScale+ MPSoC PS IP core and synthesize it in a block design.

3.2.2 MPSoC

When creating a new project in Vivado, it is important to choose the exact hardware that will later be used to upload the code onto. This is important because timings, pins, clocks, etc, has to be exactly the same on the hardware as the software. The TEB0911 used in this project has the exact project part name: "ZYNQ-UltraScale+ TEB0911-09EG-1E. SPRT PCB: REV03, REV02. (xczu9eg-ffvb1156-1-e)". This is the specific board that has to be chosen at the start of a new Vivado project. The part of the name in parentheses can be viewed as the serial number of that particular type of board.

As mentioned in Section 3.1.1 the UltraScale+ MPSoC is equipped with a *Programmable Logic* (PL) and a *Processor System* (PS). Essentially, the PL is the FPGA and the PS is the processor, of the MPSoC. Depending on the operation that should be performed, either or both of these can programmed. When operations with higher computational complexity and/or when parallel computing is needed, it is beneficial to program the PL/FPGA. However, the PS can be more beneficial to program if the operation is simpler, less computationally demanding, and/or if greater "flexability" in the coding is needed. This is because the PS runs on C/C++ code that can be more easily changed than the coding for the PL which runs on FPGA coding languages such as VHDL.

3.2.3 Vitis IDE

Programming the PS was primarily done in the software called Vitis. This is a separate IDE from Vivado, but it can be seen as an extension of Vivado. Vitis provides and easy way of writing and uploading C/C++ code to the PS. The hardware design made in Vivado is packaged and exported to Vitis, so Vitis can also program the FPGA/PL in addition to the PS. This enables designs where both the PS and PL need to operate simultaneously. It is important to have the same Vitis version as the Vivado, so in this project Vitis 2019.2 was used.

3.2.4 GMSL2 FMC card

Choosing the correct model in Vivado is vital for the GMSL2 FMC card (TEF0010) as well. However, the FMC cards do not exist as full boards to choose in Vivado, so the correct FPGA that is on the card has to be chosen instead. As aforementioned, the TEF0010 has an Artix-7 FPGA, and its exact model name is: "XC7A100TFGG484(ABX1921)". Note that the parentheses are not actually a part of the name here, but rather added because only the part not within parentheses gives results in Vivado. Hence, "XC7A100TFGG484" can be used to search for this specific FPGA in the "parts" section when creating a new project in Vivado. Though, there are four different options and which one to choose depends on the speed grade of the FPGA. This particular FPGA was available with speed grades -3, -2, -1, -1LI, and -2L [45]. Though, in Vivado there were only -3, -2, -2L and -1 as options. The Artix-7 on this particular card (TEF0010) has a speed grade of -1 [40]. Hence, "XC7A100TFGG484-1" is the part number that correlates with the Artix 7 FPGA on the TEF0010.

3.3 Vivado: Programming an MPSoC

The first steps of programming an FPGA/MPSoC starts in Vivado. As mentioned previously, the correct board or part has to be chosen when creating a new project in Vivado, so that it matches the hardware being used. When the correct hardware has been chosen and a project in Vivado has been created, it is time to develop the design/programming. Note that a majority of the RTL designs in this project is based on IP cores and subsystems.

3.3.1 HDL wrapper

When a new project is created, the RTL needs to be designed. When the RTL design is finished an HDL wrapper needs to be created. The HDL wrapper is generated based on the RTL design, and consist of VHDL or Verilog code that describes the RTL design in its entirety. The language that the HDL wrapper is generated in depends on the target language chosen in the project settings. This project will exclusively use VHDL as the programming language, since all three thesis members have previous knowledge and experience in this language.

3.3.2 Synthesis, Implementation, and Constraints

An HDL wrapper is mainly needed so that the design can be synthesised and implemented. *Synthesis* effectively transforms the HDL code "...into a synthesized netlist of library cells..." [46]. This synthesized netlist can in turn be used for *Implementation* in Vivado. An Implementation is described by Xilinx as a "...placement and routing tool for AMD devices, generating bitstreams and device images..." [47]. When the Implementation is finished an *Implemented Design* is generated where the full structure of the targeted FPGA can be seen, with all its pins and connections specified to these pins. How the pins of the FPGA are connected can either be specified before Synthesis and Implementation in a so called *Contraints File* (XDC), or directly in the Implemented Design. This is often called *port mapping*, which effectively describes how the inputs and outputs of the RTL design (software) is connected to the pins of the FPGA (hardware). The constraints file is written in its own language called XDC, hence the constraints file can also be referred to as an XDC-file [48]. Changing the port mapping directly in the Implemented Design instead, allows for a more interactive way of port mapping where all the pins and their information is readily available. Refer to Section 3.7.3 and Figure 3.28 for a concrete example of port mapping in the Implemented Design. There is also a tool in the Implemented Design called *Constraints Wizard*, that can help the user define constraints in a more user-friendly manner, compared to defining and writing them directly into the XDC/Constraints file.

3.3.3 Bitstream and Hardware Specification

When the design is synthesized and implemented, a bitstream can be generated that describes the entire system with all connections, timings, voltage levels, etc. The bitstream is essentially the file that is used to upload the code onto the FPGA. Hence, the design/programming process is finished here if only the FPGA/PL of the MPSoC is to be programmed. However, if there is a need to program the PS as well, the project needs to be exported to the Vitis IDE. This is done by exporting *Hardware Specifications* from Vivado. During the export, there is an option to include the bitstream in the Hardware Specification. This is important to remember so that the code can be uploaded to the FPGA within the Vitis IDE.

3.3.4 Vitis IDE and Hardware Manager

In the Vitis IDE an *Application Project* is made, based on the Hardware Specification file with the included bitstream. In this Application Project multiple C/C++ code files can be added, written or edited depending on what the user want the PS to execute. When all relevant code is included, the next step is to `build project`. `Build project` is an executable action in Vitis which "...builds the board support package...", "...compiles the application software using platform-specific gcc/g++ compiler...", and links the application software to the board support package [49, p. 45]. This can be seen as a regular C compiler, but where the software and the hardware is compiled compiled together. There are two main options for the `build project`; `Debug` and `Release`. This project exclusively uses `Debug`, as it is more beneficial during development and testing. When the build is finished the project can be uploaded to the hardware. There are multiple options for this, but the main way used in this project is right-clicking on the project in the file explorer and choosing the executable option `Debug As` → `Launch on Hardware (Single Application Debug)`. If no errors appear, the code is uploaded to both the PS and the PL and the MPSoC is thereby fully programmed.

An alternative to uploading the code in Vitis is to use the Hardware Manager in Vivado. When only the PL/FPGA needs to be programmed and not the PS, the bitstream can instead be uploaded directly via the Hardware Manager to the FPGA. The first step in the Hardware Manager is to open a new target. When connected to the FPGA via JTAG from the PC, the FPGA will show up as an available "Hardware Target". Choosing the FPGA opens up several windows that give the user multiple options on how to interact with the FPGA. Options such as directly upload code/bitstream directly to the FPGA, live mointoring of certain signals in a waveform window, live configuration of certain signals, and more. For details regarding monitoring and configuration of live signals on the FPGA, refer to sections 3.7.4 - 3.7.7.

3.4 CSI2 Tx Subsystem Specifics

The main purpose of this degree project is to investigate how to handle the input and output of the FPGA on the GMSL2 FMC card so that the serializisers on the card get data a clocks on the CSI2 protocol format as inputs. To do so, code is needed that can convert the AXI4 stream of data, that comes from the MPSoC, into CSI2. There are essentially two options here: write a VHDL file that can convert AXI to CSI2 and that follows both protocols specifications and timings exactly, or take an already developed subsystem by Xilinx that only needs to be configured to the specific requirements of the project. The latter was chosen to save time and effort.

Effective from Vivado version 2020.1, the so called *MIPI CSI Controller Subsystem* is bundled with Vivado. This means that all versions after 2020.1 had the license for the *MIPI CSI Controller Subsystem.* This includes both a *MIPI CSI2 Rx Subsystem* and a *MIPI CSI2 Tx Subsystem* [50]. Here, the Tx subsystem is what is needed, since the aim is to transmit (Tx) CSI2 signals, not receive (Rx). However, due to the 2019.2 version being used, the license for the *MIPI CSI2 Tx Subsystem* had to be requested via Xilinx and added to Vivado externally. Though, the license acquired is only for hardware evaluation/testing and is only active for 120 days. Therefore, it can not be used for longer period of times on the hardware but it works fine for this project. For brevity, this subsystem will often be refered to as the "CSI2 Tx Subsystem" or the "Tx Subsystem". In Figure 3.13 the CSI2 Tx Subsystem can be seen, as displayed in the block design of Vivado.

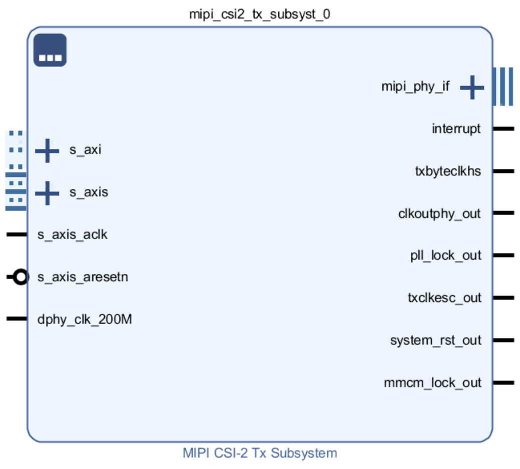


Figure 3.13: MIPI CSI-2 Tx Subsystem IP

An expanded version of the MIPI CSI2 Tx Subsystem can be observed in Figure 3.14. On the input side (left) the interfaces of the AXI4-Lite (`s_axi`) and AXI4 (`s_axis`) protocols can be seen expanded. The purpose of the protocols have already been explained in the Section 2, but the AXI4-Lite essentially controls the initialization of the subsystem while the AXI4 handles the receiving of image data. There are also two clocks as inputs: `s_axis_aclk` is the AXI reference clock which should be the same for all parts of the design that are AXI-interconnected, and `dphy_clk_200M` is the 200 *MHz* clock used for the Tx subsystem's DPHY control. The DPHY clock pin should be connected to a dedicated 200 MHz clock [51]. It is also important to note that the system clock used for most of the project was a 100 *MHz* clock.

On the output side (right) the interface for the CSI2 protocol can be seen with pin name `mipi_phy_if`. This interface includes the differential pairs of the CSI2 data lanes and the clock lanes. Depending on the IP configuration more data lanes can be added. If the CSI2 lanes are changed to e.g 4 instead of 1, `mipi_phy_if_data_n[0:0]` would instead be a bus called `mipi_phy_if_data_n[3:0]` with 4 CSI2 lanes. The "p" and "n" in the names for the data and clock lanes signifies whether the signal, for the differential signal pair, is a non-inverted (positive) or inverted (negative) signal, respectively. The `interrupt` pin is used for generating interrupt to indicate status information. Pins `txbyteclkhs - mmcm_lock_out` are all used for connecting additional subsystem to the output stage of the Tx Subsystem [51]. Though, this is not relevant for this project.

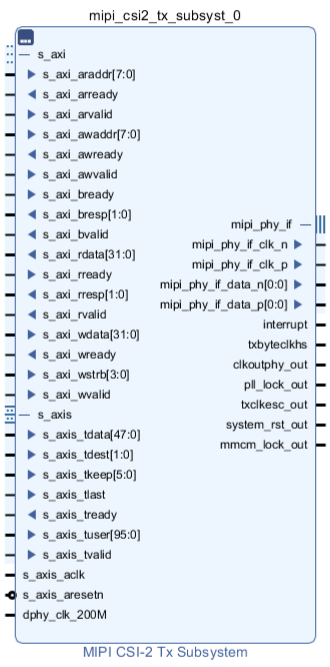
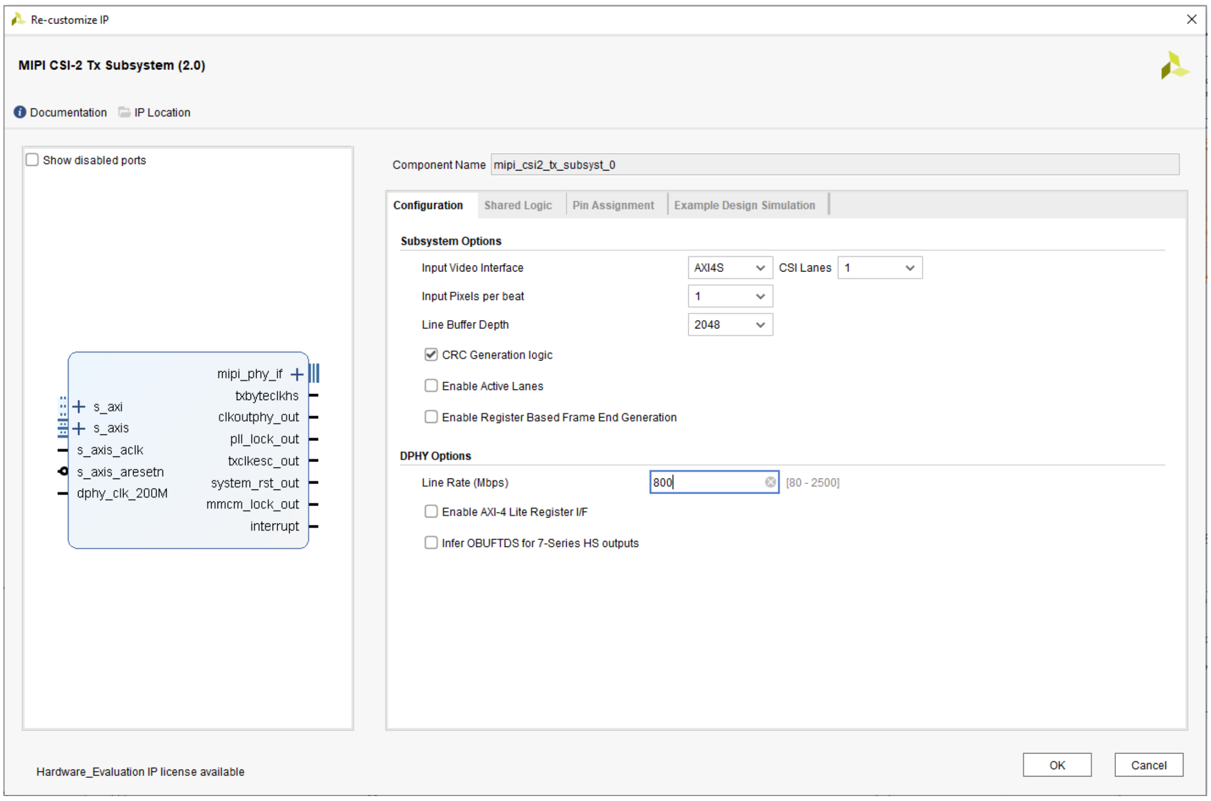


Figure 3.14: MIPI CSI-2 Tx Subsystem IP Expanded

Additionally, in Figure 3.15 an example is displayed of how a the configuration GUI for the IPs can look. This specific window is for the CSI2 Tx Subsystem, and all the adjustable parameters are specific for this IP. But most IPs have a similar window with different parameters that are easily adjusted.



MIPI CSI-2 Tx Subsystem (2.0)

Component Name mipi_csi2_tx_subsyst_0

Subsystem Options

Input Video Interface AXI4S CSI Lanes 1

Input Pixels per beat 1

Line Buffer Depth 2048

[✓] CRC Generation logic

[ ] Enable Active Lanes

[ ] Enable Register Based Frame End Generation

DPHY Options

Line Rate (Mbps) 800 [80 - 2500]

[ ] Enable AXI-4 Lite Register I/F

[ ] Infer OBUFTDS for 7-Series HS outputs

Hardware_Evaluation IP license available

Figure 3.15: MIPI CSI-2 Tx Subsystem IP Configuration GUI

3.5 CSI2 Tx Subsystem Example Design

Most IP cores and subsystems have what is called an example design. This is a Vivado block design made by the developer of the IP, that will give the user an example of how to use and build a design around the IP. This design can be generated in Vivado by adding an IP core/subsystem into a block design, right-clicking on it, and choosing "`Open IP Example Design...`". This generates a new Vivado project with the full example design and everything that comes with the design, e.g could have testbenches and other files included.

The CSI2 Tx Subsystem IP has an associated example design that can be seen in Figure A.1.1 under Appendix A.1.1. This design is constructed with multiple different IP cores and subsystems. Each core/subsystem presented in the example design will be described in detail in sections 3.5.1 - 3.5.6 . All figures shown of the IP's in these sections are directly pulled from the full example design, as seen in Figure A.1.1. This means that the configurations of the IP's shown are adapted for the example design, and do not necessarily look like this initially when added to a new block design.

Furthemore, it is important to note that the example design for the CSI2 Tx Subsystem is exclusively designed and meant for simulations, and not for synthesis and implementation. However, the example design and its simulations can be used as a foundation of how to use the IP and also as a verification/proof that other, synthesizable designs, are behaving as expected.

The example design provided by Vivado have two different options. First option can generate a design with a single CSI2 data lane, RGB888 data type, and single pixel mode. Second option can generate a design with four CSI2 data lanes, YUV422 8-bit data type, and with quad pixel mode [51, p. 41]. The example design aimed at RGB888 was chosen since it seemed like the more simple option, with only one data lane and the frequently used RGB888 data format.

3.5.1 Clocking Wizard

The *Clocking Wizard* IP can be seen in Figure 3.16. The purpose of this IP is to provide an HDL source code wrapper that converts an input clock (`clk_in1`) into multiple output clocks (`clk_out1, clk_out2, ..., clk_out7`) with user defined frequencies. In addition, it also provides a timing parameter summary for the circuit using Xilinx timing tools [52]. This makes the clocking wizard a very suitable IP to use in the first stages of most block designs, since it can provide seven different output clocks with easily customizable frequencies based on one input clock. Note that the clocking wizard is connected directly to the external ports `reset_rtl_0` and `clk_100MHz`. These external ports are the global inputs of the full RTL design. For synthesis and implementation these ports have to be port mapped/connected to a physical pin on the FPGA. For simulations these can be controlled with either a testbench or directly in the waveform window. More information regarding simulations can be found in Section 3.6.

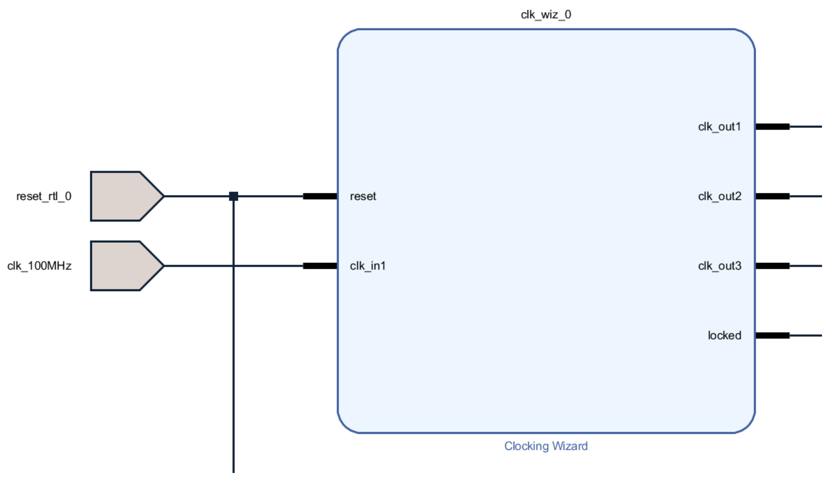


Figure 3.16: Clocking Wizard IP

3.5.2 Processor System Reset

The *Processor System Reset* IP can be seen in Figure 3.17. This IP provides customizable resets for an entire processor system, including peripherals, interconnects, and processor. It essentially allows users to "...tailor their designs to suit their application by setting certain parameters to enable/disable features." [53, p. 4]. Hence, this IP is also often included in the first stage of most block designs to centralize all resets for the block design in one place. Making the resets easier to control and customize.

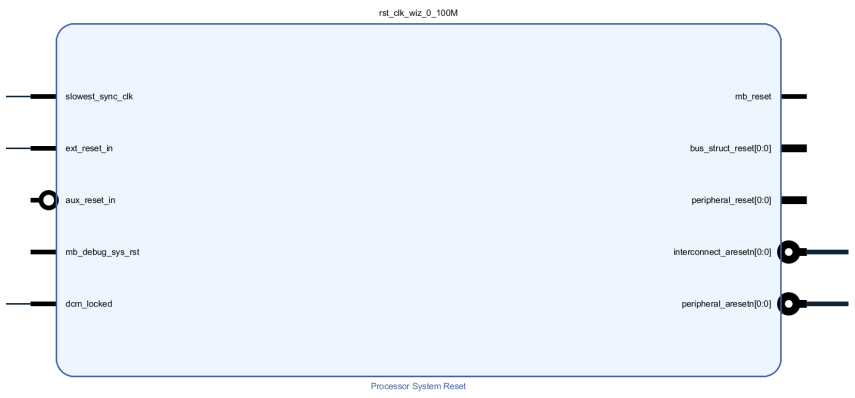


Figure 3.17: Processor System Reset IP

3.5.3 AXI Traffic Generator

The *AXI Traffic Generator* (ATG) IP can be seen in Figure 3.18. This IP provides the user a way to generate traffic for AXI4, AXI4-Stream, and other AXI4 peripherals. It has multiple different configurations that allow different AXI4 traffic generation [54]. In this example design the ATG is set to custom AXI4-Lite traffic generation with mode `System Test`. This means that the ATG can read/write AXI4-Lite transactions and checker logic. Hence, its main purpose is to act as a AXI4-Lite master, where it generates AXI4-Lite traffic to initialize and test certain IP cores in the block design. The generated AXI4-Lite traffic is sent via the output `M_AXI_LITE_CH1`. The outputs `done` and `status[31:0]` provide information regarding the IP current status. The inputs are an AXI reference clock (`s_axi_aclk`) and an AXI reset (`s_axi_aresetn`).

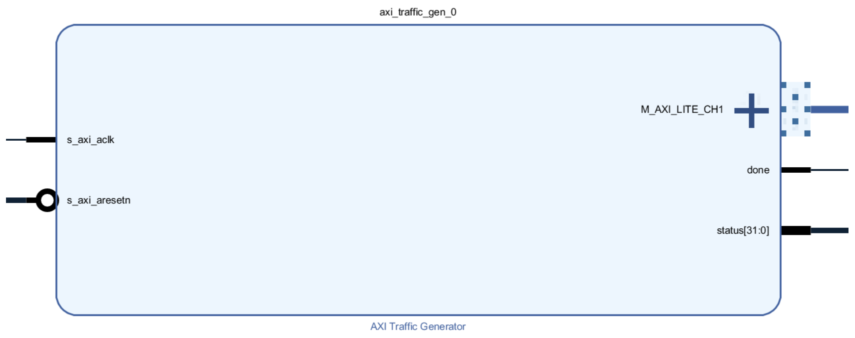
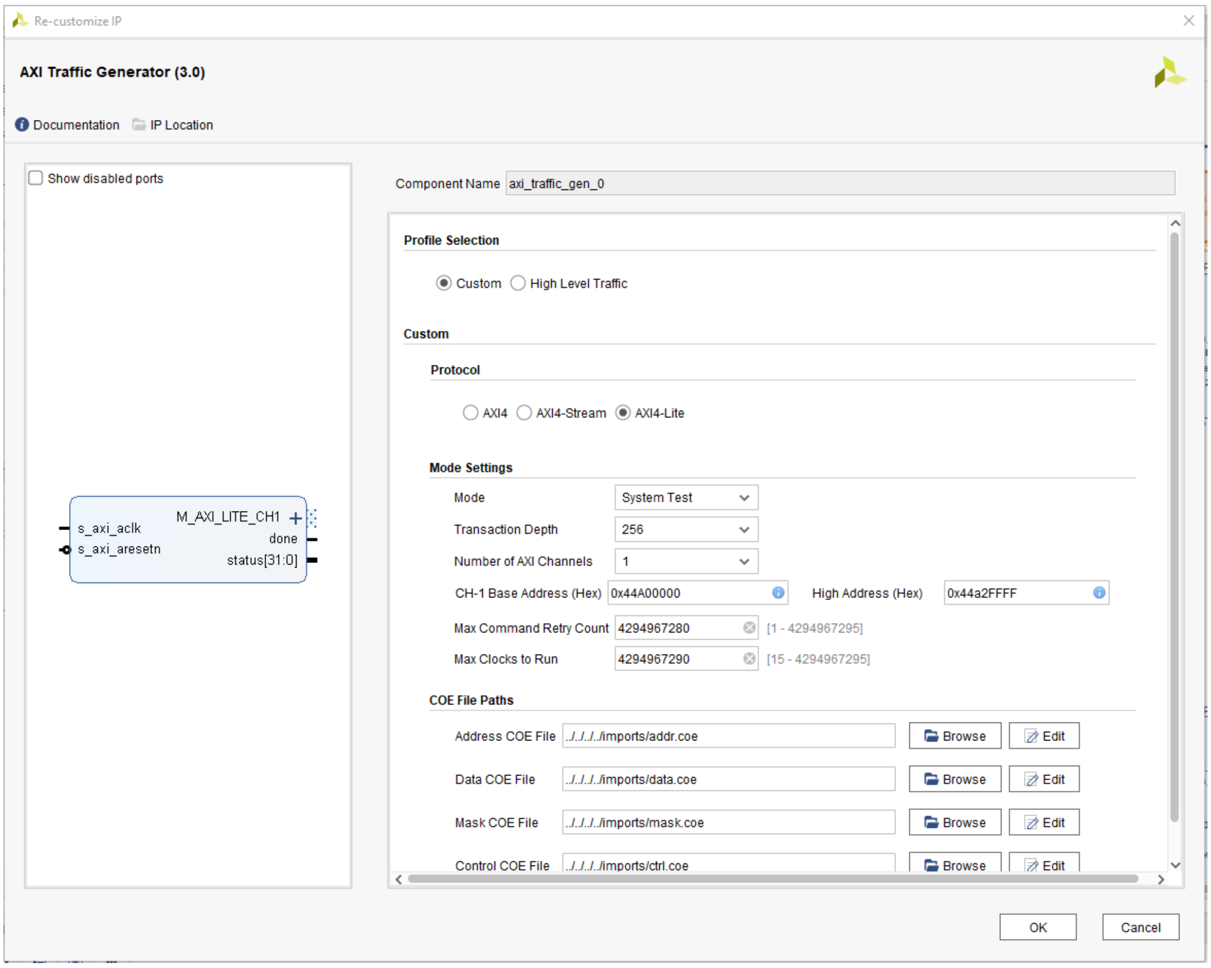


Figure 3.18: AXI Trafic Generator IP

In Figure 3.19 the configuration window for the ATG is displayed. Note that it is set to the aforementioned custom AXI4-Lite with mode `System Test`. Also note the so called *COEfficient* (COE) files. These are files needed when the core is set to "Custom". The COE file is an "...ASCII text file with a single radix header followed by several vectors" [55, p. 95]. These files are needed to describe the paramater values for some IPs. In `System Init` mode, only COE files for address and data are needed. Theses files describe the data that should be generated and to which AXI addresses. Or more precisely, the Address COE file "provides the sequence of addresses to be issued" and the Data CEO file "provides the sequence of data corresponding to the address specified in Address COE File" [54, p. 23]. Where first entry in the in the Adress COE file corresponds to first entry in the Data COE file. `System Test` mode is essentially an enhancement of the `System Init` mode that allows the core to generate read transactions [54, p. 24]. There are two additional COE files needed here; Mask and Control. The Mask COE file "...represents the bits to mask before comparing the read data versus expected data" [54, p. 24] and the Control COE file "contains the control information of type of transaction to be generated, next COE entry to fetched, and to count if any errors occurred" [54, p. 57]. The COE files are fully user programmable, so they need to be written directly by the user. Though, in this example design they are already written and provided to the user.



Component Name: axi_traffic_gen_0

Profile Selection: [✓ Custom] High Level Traffic

Custom

Protocol: AXI4 AXI4-Stream [✓ AXI4-Lite]

Mode Settings

Mode	System Test		
Transaction Depth	256		
Number of AXI Channels	1		
CH-1 Base Address (Hex)	0x44A00000	High Address (Hex)	0x44a2FFFF
Max Command Retry Count	4294967280	[1 - 4294967295]	
Max Clocks to Run	4294967290	[15 - 4294967295]	

COE File Paths

Address COE File	../../../imports/addr.coe
Data COE File	../../../imports/data.coe
Mask COE File	../../../imports/mask.coe
Control COE File	../../../imports/ctrl.coe

Figure 3.19: AXI Trafic Generator IP Configuration Window

3.5.4 AXI Interconnect

The *AXI Interconnect* IP can be seen in Figure 3.20. This IP interconnects one or mutiple AXI master devices to one or more slave devices [56]. So, in the example design the two AXI4-Lite masters `axi_traffic_gen_0` (ATG0) and `axi_traffic_gen_1` (ATG1) are connected to the inputs `S00_AXI` and `S01_AXI` of the AXI Interconnect IP. Consequently, the outputs of the IP (`M00_AXI - M02_AXI`) are connected to the three AXI slaves `mipi_csi2_tx_subsyst_0`, `v_tpg_0`, and `mipi_csi2_rx_subsyst_0`. The input clocks and resets are the clocks and resets associated with the respective master and slave.

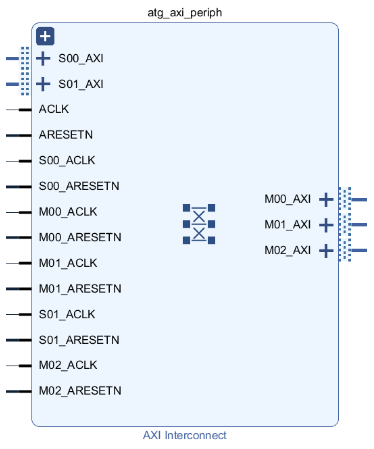


Figure 3.20: AXI Interconnect IP

3.5.5 Video Test Pattern Generator

The *Video Test Pattern Generator* (VTPG) IP can be seen in Figure 3.21. The function of this IP is to generate test patterns for video system debugging, bring up, and evaluation. The core can be customized to generate a variety of video test patterns that allow the user to asses and debug video system quality, motion, edge, and color performance [57]. This makes the VTPG a fitting IP to use whenever a test image needs to be generated. For example, as imge data input to the CSI2 Tx Subsystem. The image resolution and what type of test pattern the image should include, can be adjusted in the VTPG's configuration GUI. The input `s_axi_CTRL` takes AXI4-Lite communication. The clock (`ap_clk`) and the reset (`ap_rst_n`) are the AXI reference clock and AXI reset. The output `m_axis_video` allows tranmitting of image data via AXI4 interface. Note here that `m_axis_video_TUSER[0:0]` is not a separate signal but rather part of the `m_axis_video` interface. However, it is connected elsewhere so it is displayed as its own separate signal.

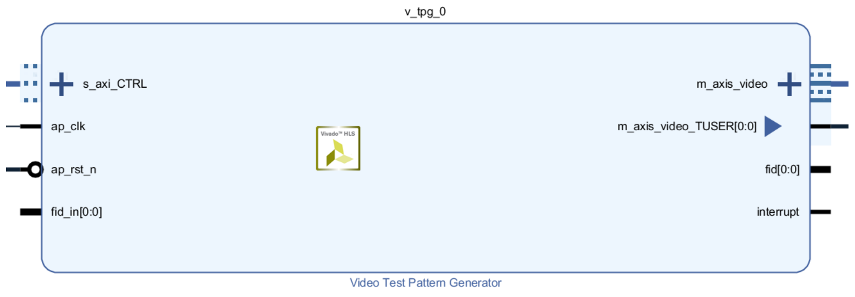


Figure 3.21: Video Test Patter Generator IP

In Figure 3.22 an expanded version of the VTPG in Figure 3.21 can be seen. Here the AXI4-Lite input and AXI4 output interfaces can be seen in detail with its respective signals. The output signal `m_axis_video_TDATA[23:0]` is important to note because this is where the image data is transmitted [57]. Additionally, there are some other important signals; `TLAST`, `TVALID`, `TREADY`, etc, that control the communication between the VTPG and the CSI2 Tx Subsystem in the example design. These will be discussed in greater detail in Section 4.1, where more concrete simulation results can be linked to their behaviour.

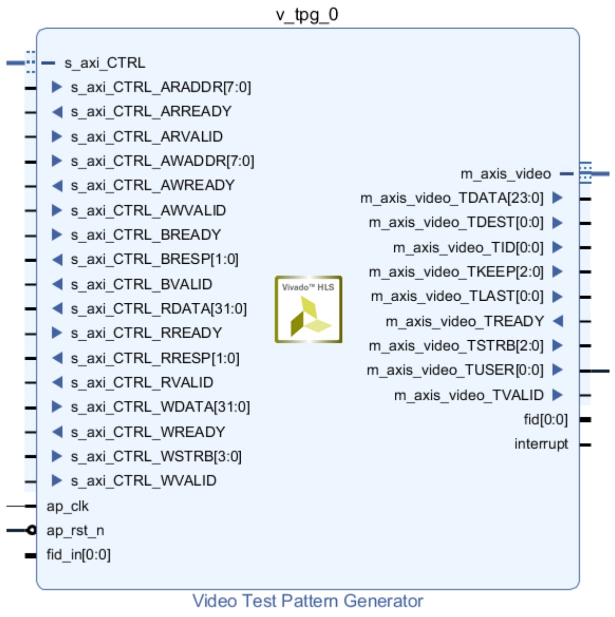


Figure 3.22: Video Test Patter Generator IP Expanded

3.5.6 AXI4-Stream Subset Converter

The *AXI4-Stream Subset Converter* can be seen in Figure 3.23. This IP provides a way to connect two separate AXI4-Streams that slightly differ in their structure. It has configurable AXI4-Stream signals for each interface that provides the user with a consistent way of converting one signal set into another. In the example design, this IP is used between the VTPG AXI4 output and the CSI2 Tx Subsystem AXI4 input as a way to match their AXI4 interfaces. The input `S_AXIS` receives the AXI4-Stream that is converted and transmitted via the output `M_AXIS`. Input signals `aclk` and `aresetn` are AXI4 reference clock and reset.

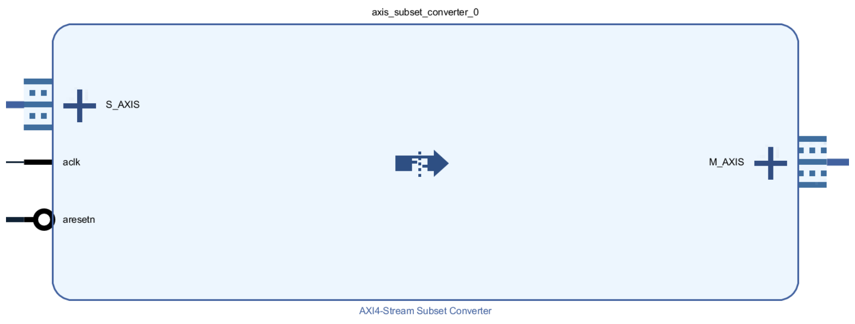


Figure 3.23: AXI4-Stream Subset Converter IP

3.5.7 MIPI CSI-2 Rx Subsystem

The *MIPI CSI2 Rx Subsystem* IP can be seen in Figure 3.24. Instead of receiving AXI4-Stream and transmitting signals on the CSI2 format, like the CSI2 Tx Subsytem does, this IP receives signals in CSI2 protocol and converts it into Native video data stream. In the example design, this IP's main function is to create a feed-back from the output of the Tx Subsystem. Allowing the user to investigate the image data output from the Tx as Native image data stream as output from the Rx Subsystem. This means, the Tx Subsystem's output port `mipi_phy_if_1` is looped back and connected to the Rx Subsystem's input port `mipi_phy_if_0`, as can be seen in the example design in Figure A.1.1. Though, it is important to note that these ports are not physically connected in the block design, but rather connected via the testbench. Refer to Section 3.6 for further information regarding testbenches. This feedback to the Rx Subsystem can be a useful tool when simulating the design, but will not be relevant for future designs that aim at synthesis and implementation. Mainly due to the fact that, for implementation, the input port can not be connected to the output port while it is connected to a physical pin. Essentially, making the connection generated in the testbench a virtual one that exclusively works for simulations. The Native video data is transmitted on the output `video_out_tdata[23:0]`. The remaining outputs will be discussed in greater detail in Section 4.1, as they are more relevant when linked to simulation results.

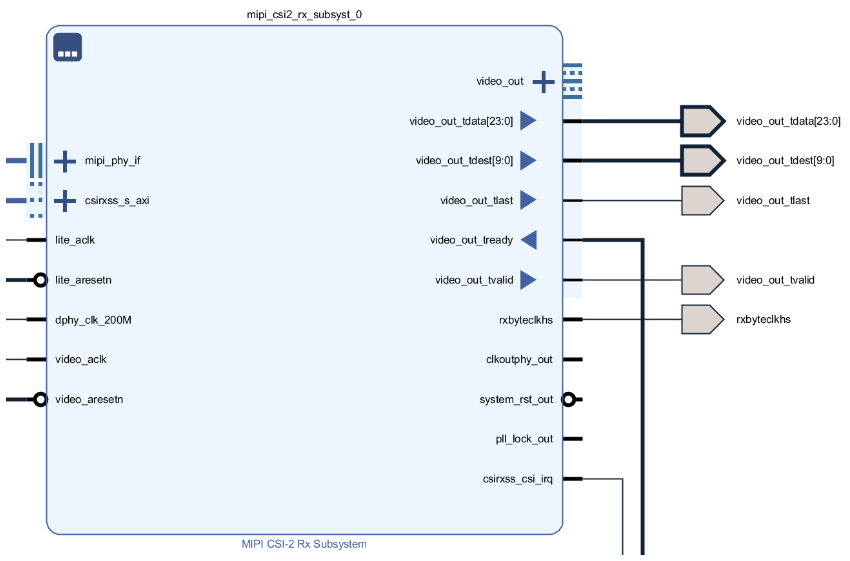


Figure 3.24: MIPI CSI-2 Rx Subsystem IP

3.6 Simulation Setup

Before simulating an RTL design, it is important to know how simulations work in the specific software, in this case Vivado, and the different ways of setting up a simulation. As explained in Section 3.3 an RTL design has to be developed in order to synthesize, implement and generate a bitstream from the design. This is the case when doing simulations as well, but the design does not have to be synthesized or implemented. Simulations is a separate executable tool that Vivado present, but it still needs a valid RTL design as input.

Vivado's simulation environment requires input signals to the external input ports. This is due to the fact that the ports are not mapped/connected to any physical pins on the FPGA when doing simulations. Hence, the user has to provide the software with input signals. This can be achieved mainly in two different ways: directly adding and configuring the signals in the waveform window (WFW) or writing a testbench (TB). Where the waveform window is the window in Vivado that displays the simulated signals in a waveform. This can be seen in greater detail under Section 4.1, where most results are presented using a WFW.

Furthermore, a testbench is a file written by the user, in either Verilog or VHDL, that specify how the input signals should behave and how some ports should be interconnected. In the example design, a testbench is provided in Verilog and should work directly with the example design. However, to really understand how the example design works, how to write a testbench in Vivado, and to get full control over the simulations, an additional one was written in VHDL for this project. The VHDL code for this TB can be seen in Appendix A.2.2.

The TB is quite a simple one, as the only inputs that need to be generated are a 100 MHz clock and a reset. Additionally, the external output port `mipi_phy_if_1` needs to be looped-back/connected to the external input port `mipi_phy_if_0` in order to enable the feedback from the Tx Subsytem to the Rx Subsystem. This connection is realized in the TB by mapping each output signal to its respective input signal.

3.7 Implementing the Software on the Hardware

Implementing the software on the hardware is the main initial problem to overcome for the project to progress further. It is also one of the more difficult stages of most embedded system implementations. This section will describe how the CSI2 Tx Subsystem Example Design was edited and what tools were used to allow the design to be implemented on the hardware.

3.7.1 TEF0010 Lead Times and Complications

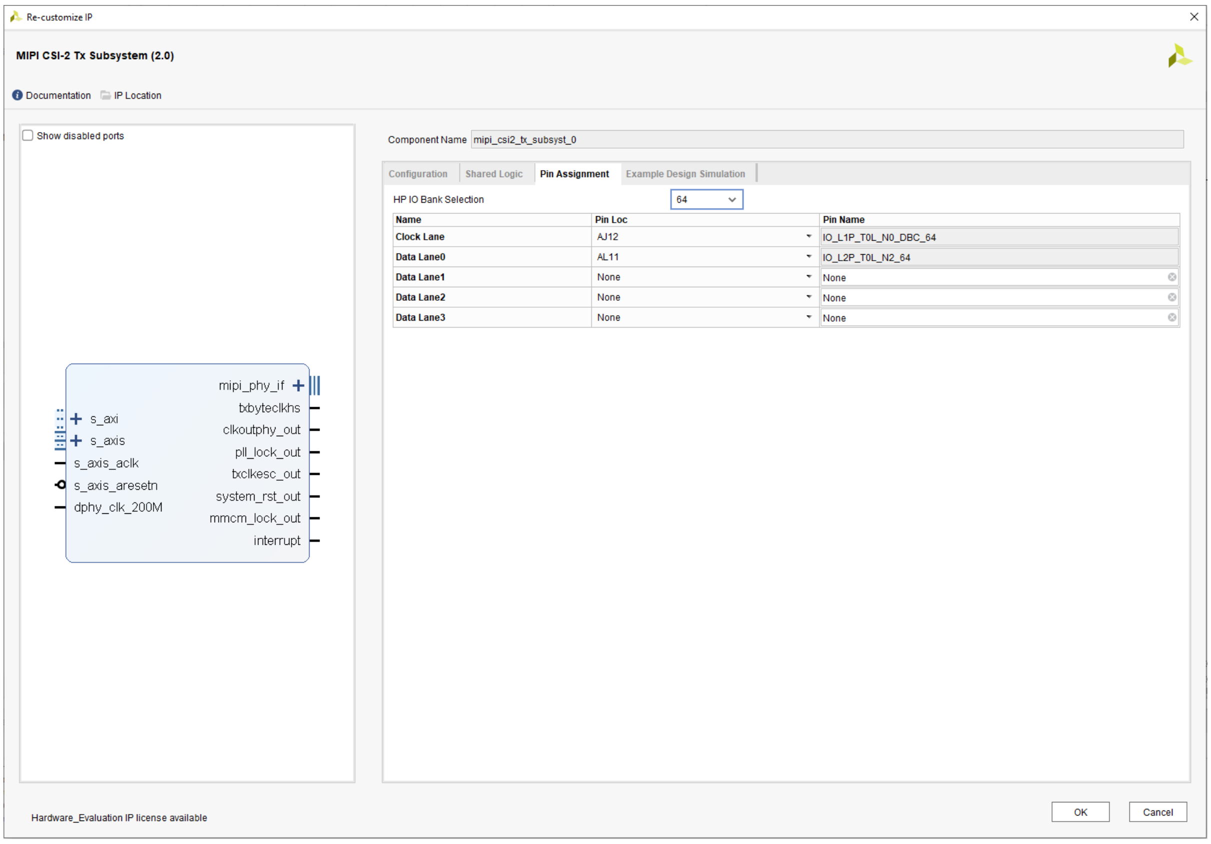
Certain parts of the project, was initially halted due to lead times on specific parts of the hardware. More specifically, the GMSL2 FMC card TEF0010 had a delivery lead time that was longer than expected. The card was delivered and mounted to the TEB0911 around week 11 of 2023. This made working on the specific hardware impossible until this time. So, the work up until this point mainly consisted of research and preparing for the card to arrive. Preparation such as creating a graphical block design of the card with important signals, port mapping between the TEF0010's FPGA and TEB0911's FPGA, software simulations, etc. When the card eventually arrived, efforts were made to program the card via Vivado's Hardware Manager, as had been proven to work with the TEF0007 FMC card, but to no avail. The FPGA that is on the card simply did not show up as an available device in the Hardware Manager. Many ideas regarding this fault were explored, including initializations, switch configurations on the TEB0911, CPLD's firmware, and more. After some weeks of attempts and extensive research regarding the cause of the problem, the card was dismounted from the TEB0911 to examine it closer. Upon inspecting the card it was found that the card was not the expected TEF0010, but rather a completely custom card designed by the company currently providing the IU solution. It did not include any Xilinix FPGA, which is why it did not show up as a device in Vivado's Hardware Manager. Rather, it is operated by two FPGAs called *CrossLink* which are produced by *Lattice Semiconductor* [58]. These FPGAs can be programmed using a different software called *Lattice Diamond Programmer* [59]. However, this was not an available option since programming the card is only possible when sufficient information regarding the hardware is available. Either through schematics or technical reference manuals. This being a fully custom card, and all schematics and technical reference manuals being confidential, made programming the hardware practically impossible.

3.7.2 Migrating the Programming to the MPSoC

Due to the unforeseen circumstances regarding the GMSL2 FMC card, the approach of the output stage had to be changed. Rather than running the code/operations on the GMSL2 card's FPGA, it was instead migrated to the MPSoC/FPGA on the TEB0911. This was done in an attempt to salvage relevant results for the thesis. I.e attempting to implement the code on the TEB0911 FPGA could still provide proof that the CSI2 Tx Subsystem could be used as a possible solution for a GMSL2 card, with hardware produced by Xilinx instead. However, since no information is available for the current GMSL2 card, no actual GMSL2 signal can be produced from the CSI2 signals that were supposed to be sent from the FPGA on the TEF0010 to the serializers on the TEF0010. So, the focus shifted toward producing CSI2 signals from the MPSoC Ultrascale+ FPGA on the TEB0911 instead.

This was problematic for the project as a whole, but also altered the approach of the design as well. Since most of the research up until this point was aimed at the TEF0010, there was no concrete plan for how the design would be implemented on the TEB0911 instead. A lot of the work, following the discovery, went into research and development of the design so that it could be implemented on the hardware.

Firstly, the pin mapping of the outputs had to be adjusted. Though, this was quite a painless process since changing the targeted hardware in Vivado to the TEB0911 actually provided a new configuration tab in the configuration GUI for the CSI2 Tx Subsystem. This tab called *Pin Assignment*, as can be seen in Figure 3.13, provides the user a simple way of connecting/mapping the CSI2 outputs (data lanes and clock lane) of the CSI2 Tx Subsystem to pins on the TEB0911's FPGA. The port mapping is thereby realized without having to manually specify it in the *Implemented Design* or in a constraints file, as previously mentioned in Section 3.3.2 .



Component Name mipi_csi2_tx_subsyst_0

HP IO Bank Selection 64

Name	Pin Loc	Pin Name
Clock Lane	AJ12	IO_L1P_T0L_N0_DBC_64
Data Lane0	AL11	IO_L2P_T0L_N2_64
Data Lane1	None	None
Data Lane2	None	None
Data Lane3	None	None

Hardware_Evaluation IP license available

Figure 3.25: MIPI CSI-2 Tx Subsystem IP Configuration Window with Pin Assignment tab open

Secondly, the pin mapping of the inputs also had to be reconsidered. Originally the plan was to to implement the example design, or a slightly edited version of the example design, and then a custom design on the TEF0010. Since the example design only requires an input clock and a reset, these could be generated/fetched from the TEB0911 and sent to the FPGA on the TEF0010 for preliminary testing of the design. Though, as the design had to be migrated to the TEB0911, new input sources had to be found. Usually, there are multiple available clocks on SOMs that can be utilized by the user in designs. The same applies for the TEB0911, but these clocks are generated by a programmable phase-lock loop (PLL) clock generator, based on a reference clock from an oscillator, and then transmitted to pins on the FPGA. Since PLLs have to be programmed it would take some time to learn how to use the PLL. So, alternative approaches were desirable.

3.7.3 UltraScale+ MPSoC PS in Vivado

This is where the PS of the UltraScale+ MPSoC came in handy. The PS can actually be implemented into the block design in Vivado by using an IP core representation of the PS called *Zynq UltraScale+ MPSoC*. More precisely, "the IP core is the software interface around the Zynq UltraScale+ Processing System" [60, p. 4]. The IP core can be seen in Figure 3.26. Note that from this point forward in the report, the figures presented in Section 3 that show any picture from Vivado will be related to either a fully custom design or a customized version of the example design, since changes had to be made to the original example design in order to implement it on the hardware.

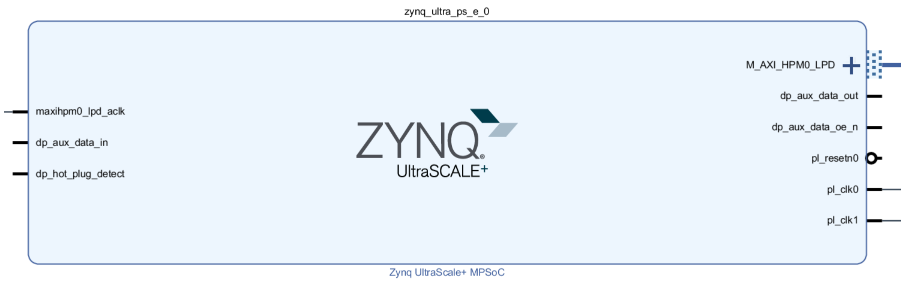
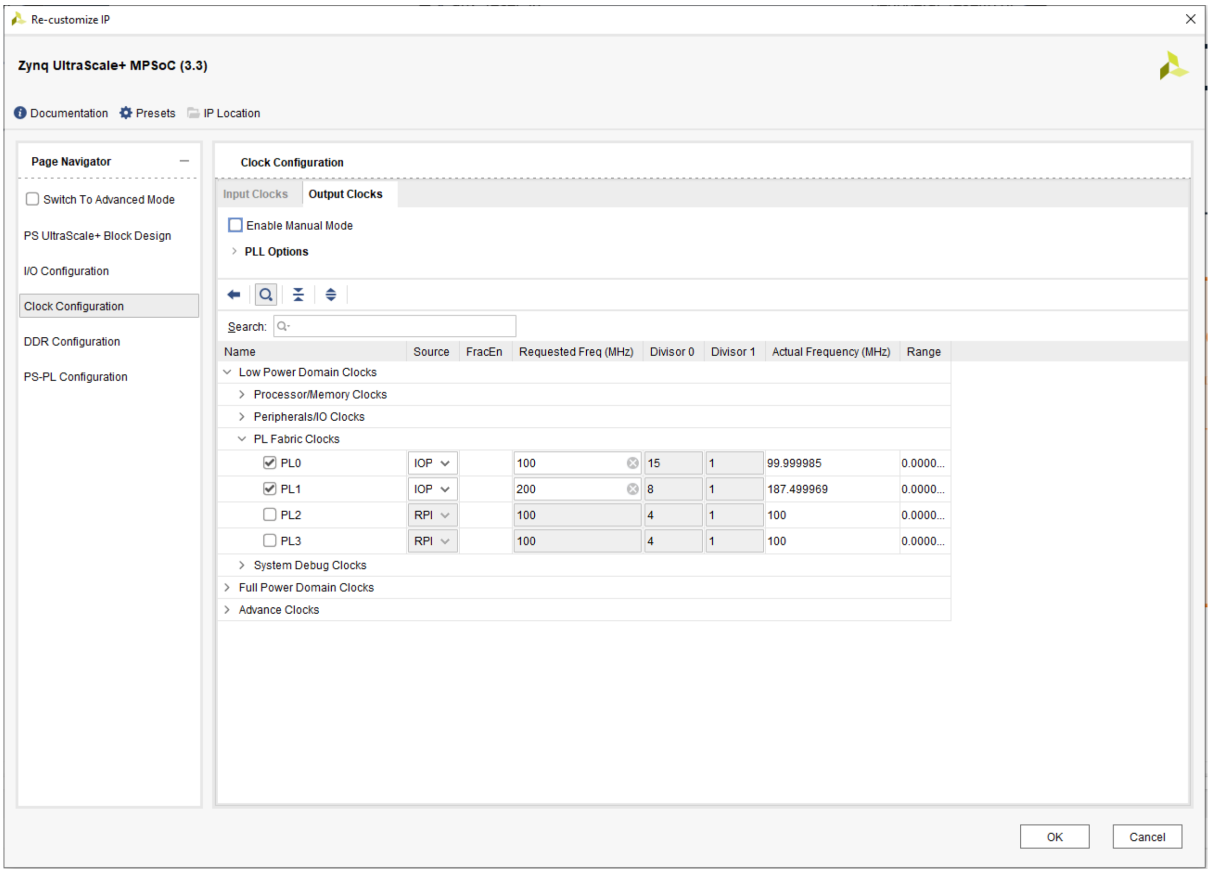


Figure 3.26: Zynq UltraScale+ MPSoC IP

Essentially this IP core allows the user to configure multiple parameters inside the PS, and gives the user a simple way of connecting and using the PS in hardware RTL designs. Among many other functionalities the IP core, more importantly for the current problem, can provide four separate PL clocks that can be used in the block design. These four PL clocks are fully customizable in terms of frequency [60]. This can be seen in Figure 3.27 where the GUI configuration tab for the PL clocks are presented. Simply enable the number of PL clocks desired and input the required frequency for each clock. Consequentially, the PL clocks show up as outputs in the IP core, as can be seen in Figure 3.26 where `pl_clk0` and `pl_clk1` are the PL clocks. This solves the problem of input clock source.


Re-customize IP

Zynq UltraScale+ MPSoC (3.3)

Documentation Presets IP Location

Page Navigator
- [ ] Switch To Advanced Mode
- PS UltraScale+ Block Design
- I/O Configuration
- Clock Configuration
- DDR Configuration
- PS-PL Configuration

Clock Configuration

Input Clocks | Output Clocks

- [ ] Enable Manual Mode
- PLL Options

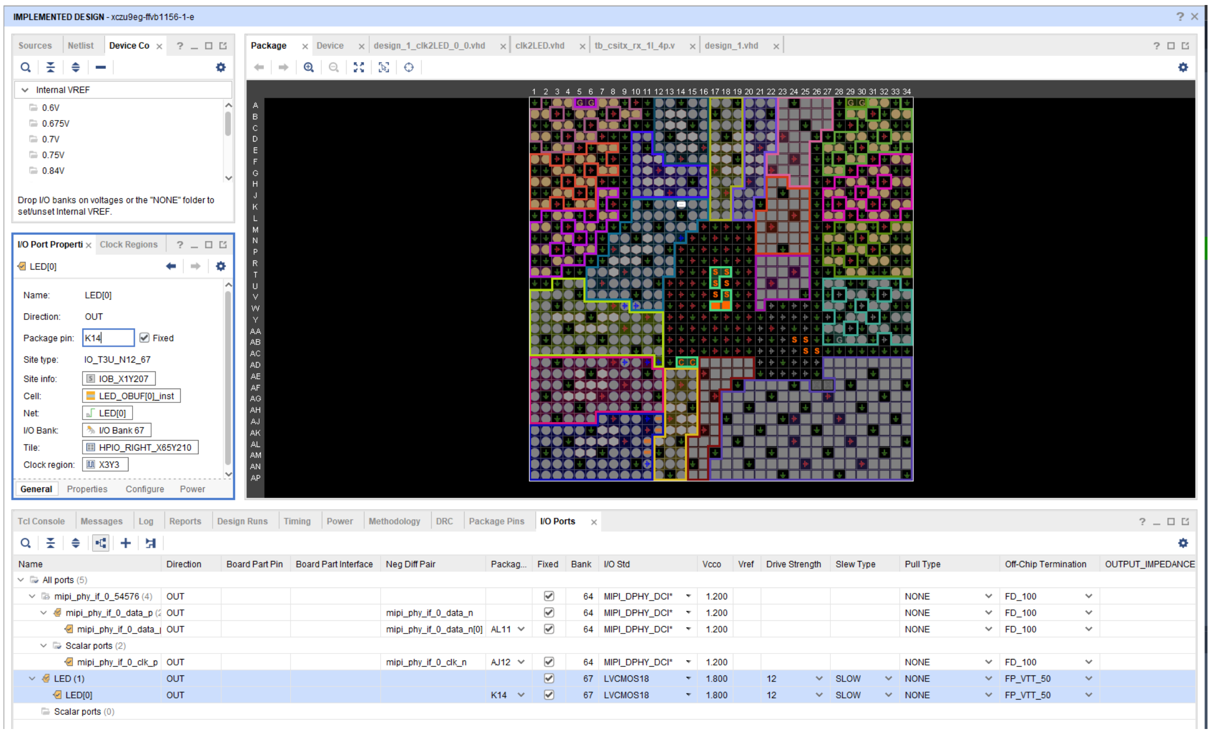
Search:

Name	Source	FracEn	Requested Freq (MHz)	Divisor 0	Divisor 1	Actual Frequency (MHz)	Range
Low Power Domain Clocks							
Processor/Memory Clocks							
Peripherals/IO Clocks							
PL Fabric Clocks							
✓ PL0	IOP		100	15	1	99.999985	0.0000...
✓ PL1	IOP		200	8	1	187.499969	0.0000...
☐ PL2	RPI		100	4	1	100	0.0000...
☐ PL3	RPI		100	4	1	100	0.0000...
System Debug Clocks							
Full Power Domain Clocks							
Advance Clocks							

OK Cancel

Figure 3.27: Zynq UltraScale+ MPSoC IP Configuration Window with PL Clock Configuration Tab

However, another problem spurred from the use of the PS IP in the block design. When the code/program was uploaded to the FPGA nothing seemed to happen on the TEB0911. This could be identified since a custom VHDL script had been written that converts the 100 MHz clock, that nearly the whole design operates based on, into a slower 1Hz clock signal. Refer to Appendix A.2.1 for this custom VHDL code, and Figure A.1.2 and A.1.3 for the RTL block called "clk2LED_0" which represent this VHDL code in the block designs. The output signal of this RTL block was in turn connected to an output port LED[0:0] that was port mapped to a LED on the TEB0911. The specific LED used in this case was a user definable LED connected to the K14 pin of the FPGA. The port mapping was specified in the Implemented Design and can be seen illustrated in Figure 3.28 below. If the design, or at least the clock, worked correctly this LED would have been blinking with 1s intervals, which it did not.



Name: LED[0]
Direction: OUT
Package pin: K14 ✓ Fixed
Site type: IO_T3U_N12_67
Site info: IOB_X1Y207
Cell: LED_OBUF[0]_inst
Net: LED[0]
I/O Bank: I/O Bank 67
Tile: HPIO_RIGHT_X65Y210
Clock region: X3Y3

Name	Direction	Board Part Pin	Board Part Interface	Neg Diff Pair	Packag...	Fixed	Bank	I/O Std	Vcco	Vref	Drive Strength	Slew Type	Pull Type	Off-Chip Termination	OUTPUT_IMPEDANCE
All ports (5)															
mipi_phy_if_0_54576 (4)	OUT					✓	64	MIPI_DPHY_DCI*	1.200				NONE	FD_100	
mipi_phy_if_0_data_p (2	OUT			mipi_phy_if_0_data_n		✓	64	MIPI_DPHY_DCI*	1.200				NONE	FD_100	
mipi_phy_if_0_data_	OUT			mipi_phy_if_0_data_n[0]	AL11	✓	64	MIPI_DPHY_DCI*	1.200				NONE	FD_100	
Scalar ports (2)															
mipi_phy_if_0_clk_p	OUT			mipi_phy_if_0_clk_n	AJ12	✓	64	MIPI_DPHY_DCI*	1.200				NONE	FD_100	
LED (1)	OUT					✓	67	LVCMOS18	1.800		12	SLOW	NONE	FP_VTT_50	
LED[0]	OUT				K14	✓	67	LVCMOS18	1.800		12	SLOW	NONE	FP_VTT_50	
Scalar ports (0)															

Figure 3.28: Implemented Design Port Mapping Example

Additionally, another LED on the TEB0911, that is pre-programmed and dedicated to signaling whether code is currently being and has successfully been uploaded, showed that the code was uploaded successfully to the board. This LED can be found under the name D6 in TEB0911's TRM [33]. Hence, something was wrong with the implementation. After multiple tries and research it was found that the PS, which generates the PL clocks, has to be initialized in order to work on the hardware. This means that C/C++ code had to run on the PS to initialize the PS in the RTL design. This can be achieved with a simple "Hello World"-script in the Vitis IDE. When uploading this simple C code together with the Hardware Specifications (which includes the bitstream/code for the FPGA) with "Run Debug" in Vitis IDE, the PS was properly initialized and the program on the FPGA started working correctly.

3.7.4 Virtual Input/Output (VIO)

The main remaining problem was the reset input source. Since the reset preferably needs to be controlled by the user in real-time during runs on the hardware, a solution needed to be found that could support this. Vivado's *Virtual Input-Output* (VIO) IP provides a solution for this. VIOs can essentially be seen as a workaround when there are either no physical pins available, so they need to be virtually represented, or when real-time control of a certain signal is needed. Hence, the IP can give the user full live control over the inputs of the design. For example, an input (output of the VIO) port can be changed from a logical '0' to a '1' while the program is running on the hardware. An example of a VIO can be seen in Figure 3.29 where the output probe_out0[0:0] can control the input reset of the whole design.

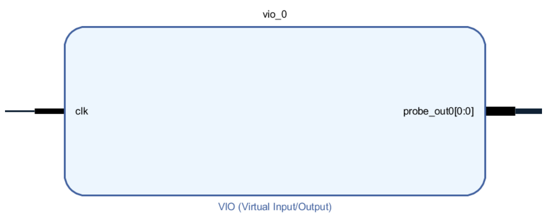


Figure 3.29: Virtial Input-Output IP as an Input Reset

Additionally, the VIO can also be used whenever there is an output port of a block design where, no physical pins are available, the signal transferred via the port is irrelevant to the implementation, or the user simply wants to actively monitor the output port. An example of this can be seen in Figure 3.30 where `probe_in0[0:0]` and `probe_in1[0:0]` are monitoring some output signals. The VIOs should run on the same clocks as the signal/operation that they are monitoring/controlling.

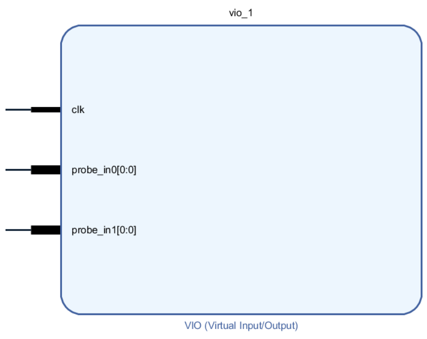


Figure 3.30: Virtual Input-Output IP as a Monitor for Output Signals

3.7.5 Integrated Logic Analyzer (ILA)

Contrary to the VIO, the Integrated Logic Analyzer (ILA) IP can be used in a similar way but for internal signals within the block design instead of external inputs/outputs. Although, the ILA can only monitor and not actively control the internal signals. The ILA is mainly used as a tool to debug and monitor designs implemented into the hardware. This makes it a useful tool whenever the programming is not behaving as expected, but no error messages are generated. By monitoring the internal signals, the user can identify where the implementation fails and make the needed adjustments. An example of an ILA can be seen in Figure 3.31 where it is monitoring multiple signals

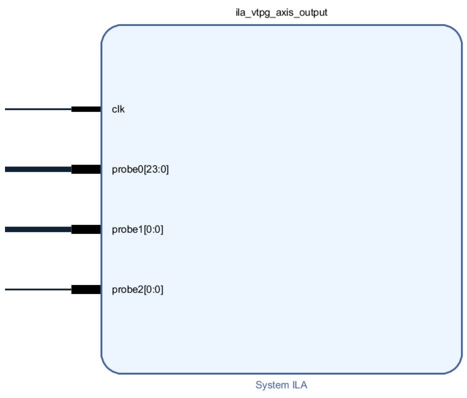


Figure 3.31: System Integrated Logic Analyzer IP

3.7.6 VIO and ILA complications

However, some problems occurred initially regarding the ILAs and VIOs during uploading. When uploading the design to the FPGA, the ILAs and VIOs were "dropped" from the design. This means that Vivado essentially omitted them. This occurred when trying to upload the program via Vivado's Hardware Manager. The error message Vivado gave mainly detailed that the ILAs and VIOs need to operate on a "free-runnning clock". This was partly due to the problem described earlier, with the initialization of the PS IP core. Since the program was first uploaded via Vivado's Hardware Manager, and not in Vitis IDE with C code to initialize the core, no PL clocks were generated. This caused the VIOs and ILAs to be dropped.

However, after changing this and uploading via Vitis with C code, the ILAs and VIOs were still dropped with Vivado reporting that they require a free-running clock. A free-running clock, according to Xilinx Forums, is a clock "...without any glitches and consistent 1-0 without skew", which "...starts it's operation automatically when the board is powered up", and is a "non-interrupted and non-gated clock" [61]. The original example design had a clocking wizard that would divide the input clock into multiple output clocks. Hence, this clocking wizard stayed in the design even after the PS IP core was added. So, one PL clock output from the PS was originally connected to the input of the clocking wizard and then divided into a number of output clocks, that could be used in the design. Though, the clocking wizard has a "locking"-function built in. This locking sequence ensures a safe clock startup, essentially causing the clocking wizard to not generate any clocks until its output pin `locked` (refer to Figure 3.16) is set to High [52, p. 43]. This is believed to be the second reason why the ILAs and VIOs were dropped. The clocks operating the ILAs, VIOs, and the entire design were not being generated straight away, causing the ILAs and VIOs to not have a free-running clock and consequently being dropped. Thereby, the clocking wizard was omitted from the design and the PL clocks from the PS IP core were used directly instead.

3.7.7 VIOs and ILAs in Hardware Manager

When VIOs and ILAs are included into a block design they are implemented into the hardware upon programming, the same as the rest of the design. However, they also show up as separate "entities" in Vivado's Hardware Manager after programming, each with their own WFW that shows the monitored signals. For results using the ILAs and VIOs in the Hardware Manager and additional information regarding the IPs, refer to Section 4.2.

3.7.8 Edited CSI2 Tx Subsystem Example Design

All IPs introduced in sections 3.7.3 - 3.7.5 were added to the original example design to enable or aid the synthesis and implementation of the design. Two main versions of the edited design are presented in the Appendix. One version of the edited design can be seen in Appendix A.1.3. Important to note that the part of the original example design that enabled the CSI2 Rx Subsystem has been removed in both designs, since that part was exclusively for simulation and could not be implemented. Also note that for this particular design the AXI traffic generators have been removed. Instead the AXI4-Lite control signals are generated from the PS IP cores output `M_AXI_HPM0_LPD`. These can therefore be controlled with C code in the PS, which provides the user with fully customizable AXI4-Lite control signals. Though, at the cost of simplicity since the C programming of AXI4-Lite signals is a complex process. There are some example codes provided by Xilinx in the Vitis IDE for certain IP cores, but these are specific for that IP core's example design. Since this design is a fully custom one, it would require fully custom C code for the AXI4-Lite signal generation. Hence, initialization of all cores in the design with AXI4-Lite from the PS has not yet been achieved.

An alternative for this is to keep the AXI traffic generators as in the original example design and not configure any of the parameters on the IP cores controlled by AXI4-Lite from the ATGs. This version of the edited example design can be seen in Figure A.1.2. Alternatively, the ATG could be configured in other ways with different modes that are not the "Custom Modes" which required COE files, as explained in Section 3.5.3. Initialization of all the cores with the ATGs have not yet been achieved. Refer to Section 4.2 for hardware tap and signal monitoring of the edited CSI2 Tx Example Design with ATGs.

4 Results and Discussion

4.1 Simulations

When working with embedded systems it can be crucial to know exactly how signals in the system are supposed to behave in theory, before the implementation process starts. This is where simulations are useful. Simulations, in this case, are exclusively software based simulations that emulate the hardware and the signals on the hardware in the form of waveforms in the waveform window. Consequently, the user can obtain tangible results that can be used as proof and a comparison of how the hardware is supposed to behave upon implementation. Note that all simulated WFW have seconds (on µs scale) as unit on the x-axis.

Due to unknown problems, simulations in Vivado 2019.2 did not work as expected for the CSI2 Tx Subsystem Example Design. More precisely, the signals stayed constant with wither '0's or '1's. Therefore, the figures below are from simulations of the example design in Vivado 2022.2 instead, since these worked as expected. Though, the block design for the 2022.2 version is nearly identical so the simulations are directly applicable to the CSI2 Tx Subsystem Example Design for the 2019.2 version.

4.1.1 CSI2 Tx Subsystem Example Design

As previously mentioned, the example design provided with the CSI2 Tx Subsystem IP includes a test-bench that specifies what signals should be generated at the external input ports and how some signals on the external ports of the design should be connected. This TB generated in Verilog was re-written in VHDL. The results of the simulations can be seen in the figures below. Note that all of the signals in the figures are color-coordinated and grouped with group names to allow easier identification regarding what protocols/interfaces the different signals belong to.

Initially refer to Figure 4.1 that shows the first instance of signal `s_axis_tready` for the AXI4 interface of the CSI2 Tx Subsystem. This signal is driven by the CSI2 Tx Subsystem and indicates that the Subsystem is ready to receive data. Note that this signal is included in the signal group "Tx Subsystem AXI Input". The signal group names are user defined group names and are only present in the WFW to make the simulations easier to interpret. The word "input" in the name "Tx Subsystem AXI Input" does not signify that all signals in this group are inputs, but rather that they are connected to the side of the IP that usually take inputs (left side). The same applies for the groups with "output" in their name. These signal groups can be found connected to the output side of their respective IPs (right side). Whether a signal is actually an input or an output can be seen to the left of the signal name in the WFW. Where inputs have a "I" in a small orange box and outputs have a "O" in a small green box. This applies for all figures presented in the results.

Additionally, observe the `m_axis_video_TREADY` of signal group "VTPG AXI Output" also goes high in the figure, since this is the input of the VTPG receiving the output `s_axis_tready` of the Tx Subsystem. Similar causations will not always be mentioned in the report because they mean little for the actual result, but it is important to be aware of.

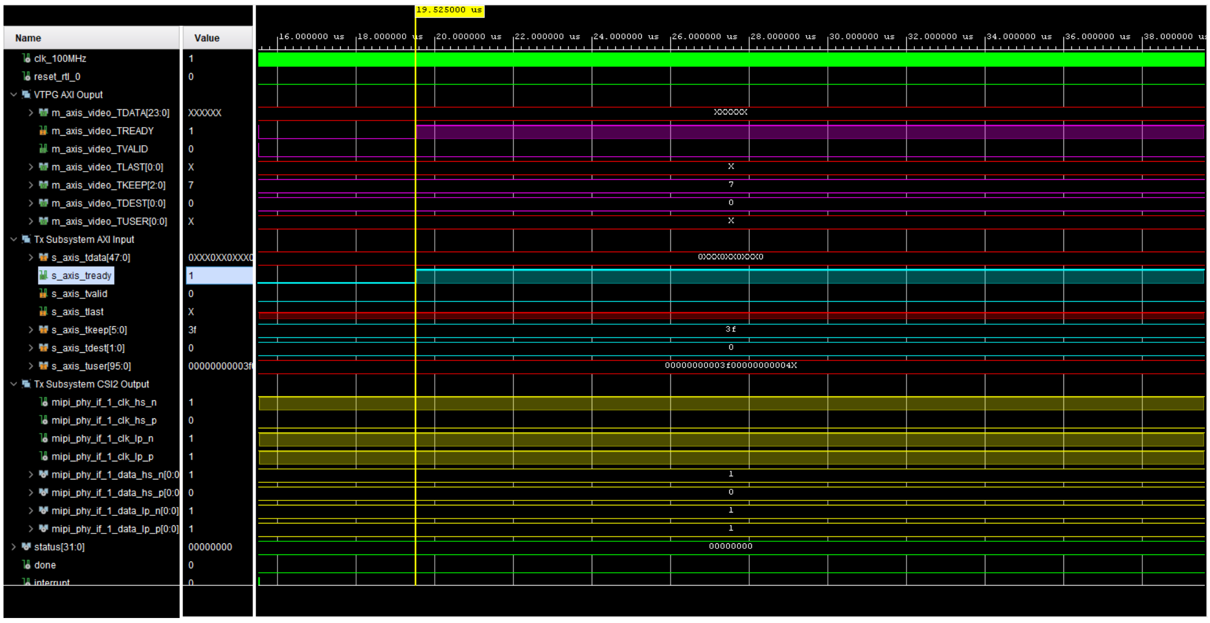


Figure 4.1: First simulations instance of `tready` signal from the CSI2 Tx Subsystem

Refer to Figure 4.2 that illustrates the beginning of actual data being transmitted. This is the first instance in the simulation that the VTPG starts sending video/image data via its output `m_axis_video_TDATA[23:0]` as highlighted in the figure. Note that it took a relatively long time (around 145 µs) for the VTPG to start sending the image data.

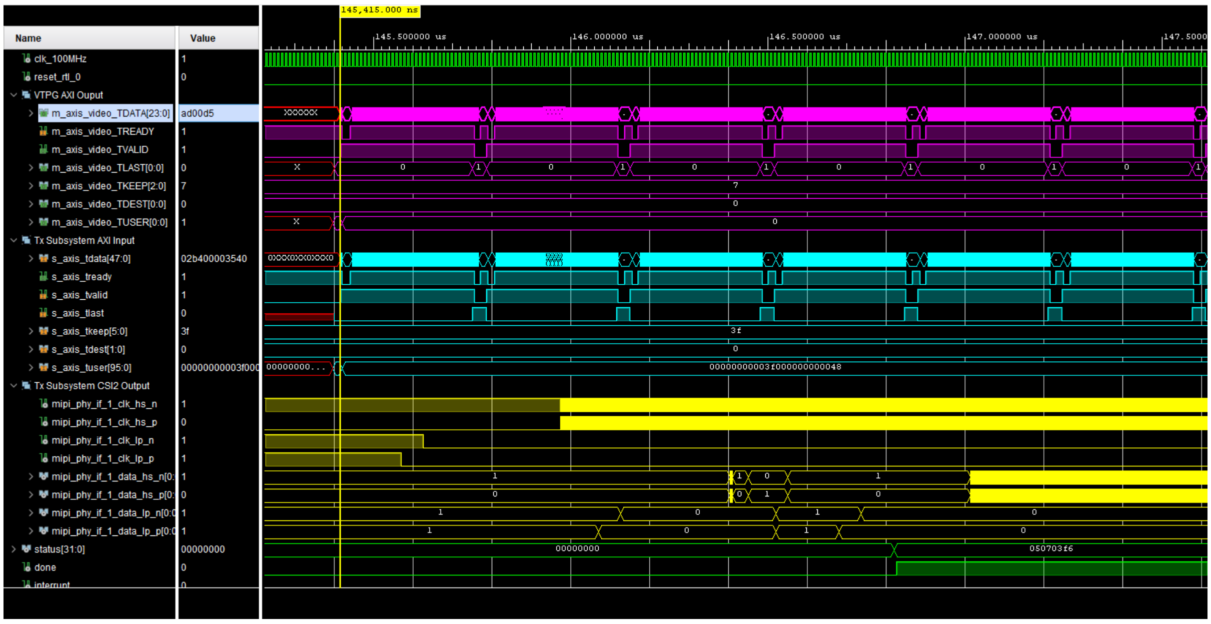


Figure 4.2: First instance of video data from the VTPG in simulation

Figure 4.3 displays the same time instance in the simulation as Figure 4.2, however it is zoomed in for greater detail to see what is actually being transmitted. Note here that `m_axis_video_TDATA[23:0]` transmits video data in "packets" of 24-bits. For example the first data packet "AD00D5" is a hexadecimal representation of the binary number "1010 1101 0000 0000 1101 0101", which is 24 bits. This is as expected since each pixel in the data frame being sent is on the RGB888 format. This means that each pixel is represented by 8 bits of red, 8 bits of green, and 8 bits of blue, resulting in a total of $8 \cdot 3 = 24$ bits. Hence, each of these data packets represent the color data for each pixel. If the image has a resolution of 64x64, which is the case for the example design, a total number of $64 \cdot 2 = 128$ data packets/pixels will be transmitted for each video frame.

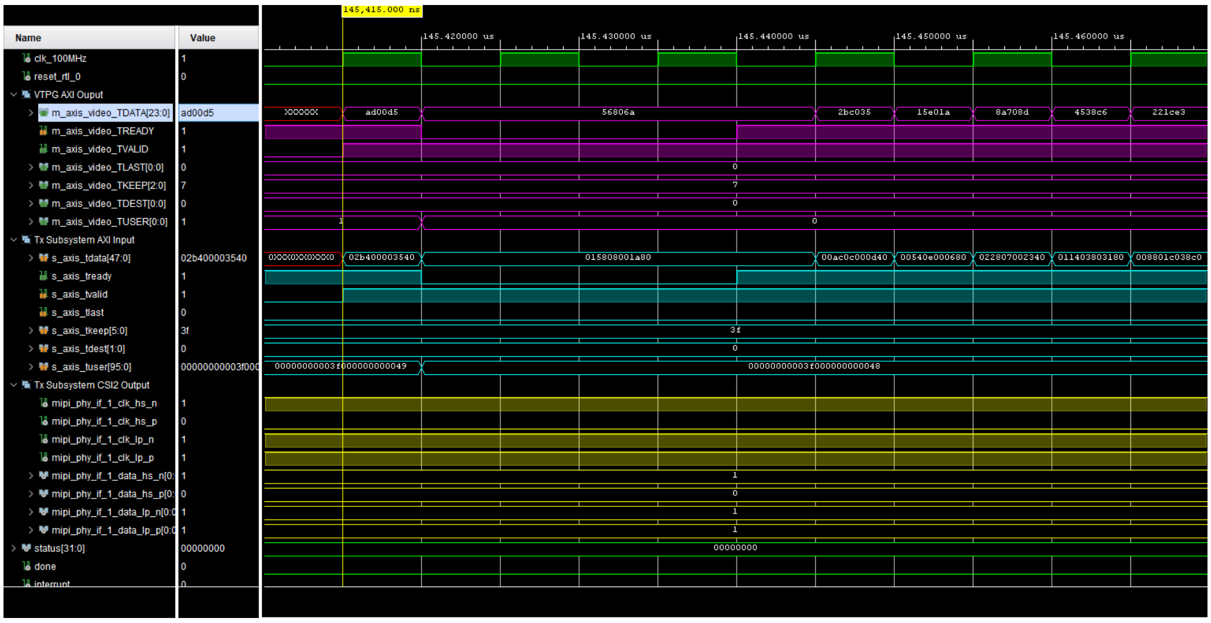


Figure 4.3: Zoomed in Figure 4.2

The reason why the image data is being sent at this particular time is because the signal `m_axis_TVALID` from the VTPG was driven high. Refer to Figure 4.4 for the highlighted signal. This signal essentially indicates that the master interface (VTPG AXI4 output) is able to provide valid samples of data to the slave interface (CSI2 Tx Subsystem AXI4 input). I.e valid video data can be transmitted from the VTPG. And since the aforementioned `m_axis_video_TREADY` is already high, which means that the CSI2 Tx Subsystem is ready to accept data, the flow of data between VTPG and Tx Subsystem can begin [57].

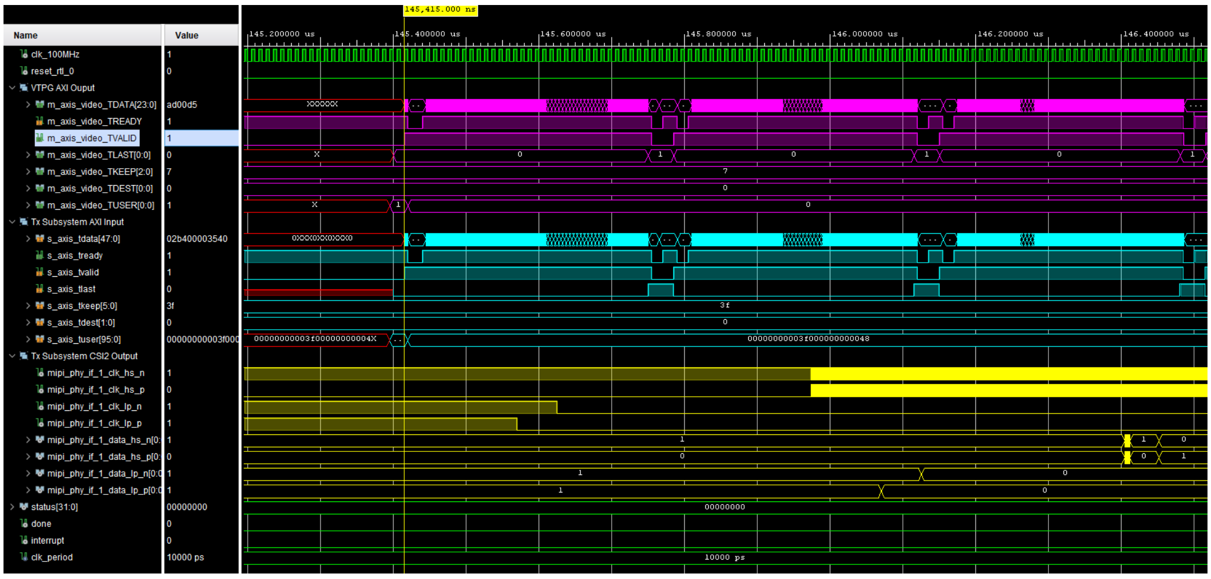


Figure 4.4: First instance of of `m_axis_TVALID`

In Figure 4.5 the first activation of signal `m_axis_video_TLAST[0:0]` can be observed. This signal signifies when the end of a line has been reached. This means, that all the pixel data has been transmitted for one line of the frame [57]. A longer pause of data transmission in `video_out_TDATA[23:0]` can be seen after `m_axis_video_TLAST[0:0]` has been asserted.

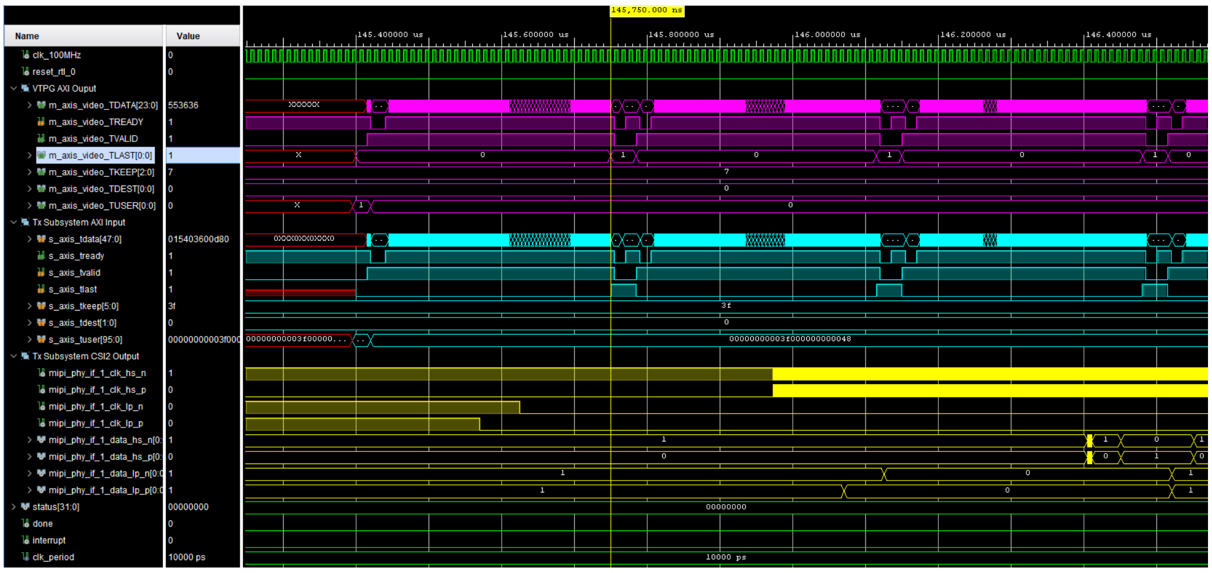


Figure 4.5: First instance of `m_axis_video_TLAST[0:0]`

A zoomed in version of Figure 4.5 can be seen in Figure 4.6. This gives a clearer look at what video data is being sent while `TLAST` is asserted. However, the product guide for the VTPG [57] does not explicitly detail what is being transferred during this time, so it is open for speculation or further research. It is most likely just buffer data that could carry information regarding line change or something similar.

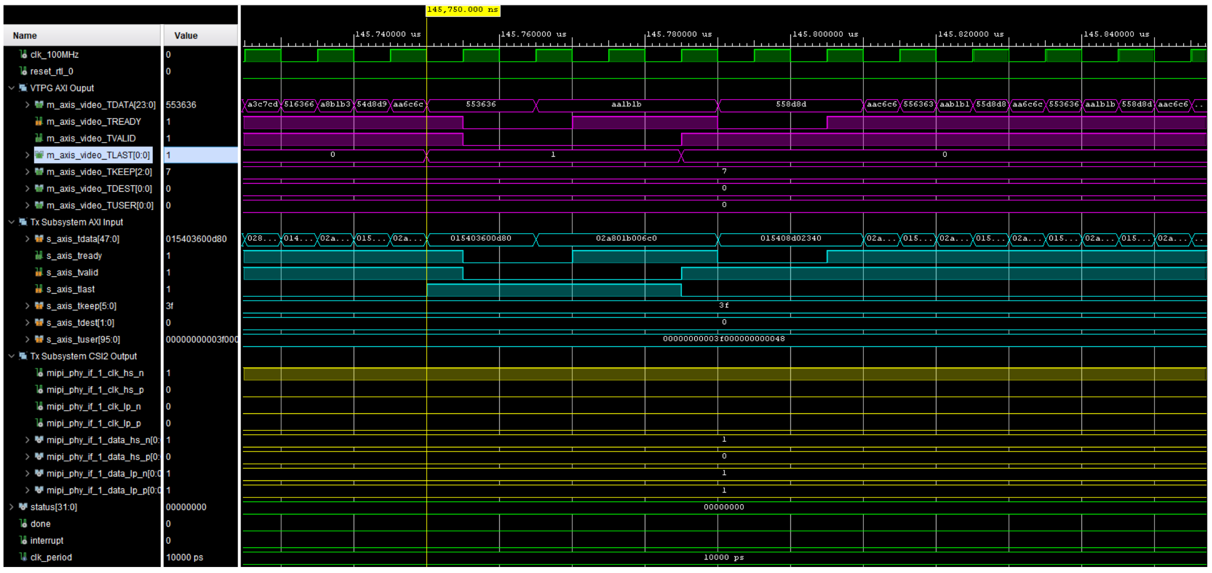
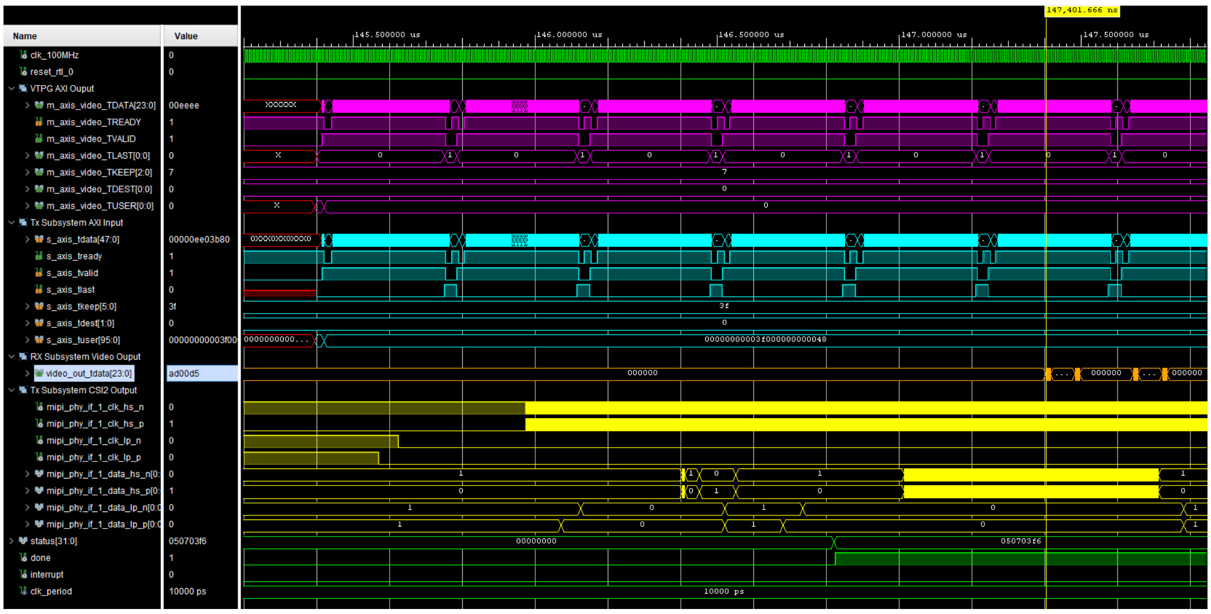


Figure 4.6: Zoomed in Figure 4.5

Additionally note that the video data transferred from the VTPG in `m_axis_video_TDATA[23:0]` is not exactly the same as the received video data to the CSI2 Tx Subsystem in `s_axis_tdata[47:0]`. This is due to the configurations of the two IPs resulting in different widths for the video data signals. The CSI2 Tx Subsystem width is for example determined by the pixel type and number of pixels per beat

[51], while the data width for the VTPG is determined through different parameters. The differences in width is handled by the AXI Subset Converter, as explained in Section 3.5.6. To prove that the video data information is preserved, the signals for the Rx Subsystem can be examined.

Refer to Figure 4.7 where the output video signal `video_out_tdata[23:0]` has been added to the WFW. This is the first instace of video data being transmitted from the CSI2 Rx Subsystem.



Name	Value
clk_100MHz	0
reset_rtl_0	0
VTPG AXI Ouput	
m_axis_video_TDATA[23:0]	00eeee
m_axis_video_TREADY	1
m_axis_video_TVALID	1
m_axis_video_TLAST[0:0]	0
m_axis_video_TKEEP[2:0]	7
m_axis_video_TDEST[0:0]	0
m_axis_video_TUSER[0:0]	0
Tx Subsystem AXI Input	
s_axis_tdata[47:0]	00000ee03b80
s_axis_tready	1
s_axis_tvalid	1
s_axis_tlast	0
s_axis_tkeep[5:0]	3f
s_axis_tdest[1:0]	0
s_axis_tuser[95:0]	00000000003f00
RX Subsystem Video Ouput	
video_out_tdata[23:0]	ad00d5
Tx Subsystem CSI2 Output	
mipi_phy_if_1_clk_hs_n	0
mipi_phy_if_1_clk_hs_p	1
mipi_phy_if_1_clk_lp_n	0
mipi_phy_if_1_clk_lp_p	0
mipi_phy_if_1_data_hs_n[0:	0
mipi_phy_if_1_data_hs_p[0:	1
mipi_phy_if_1_data_lp_n[0:0	0
mipi_phy_if_1_data_lp_p[0:0	0
status[31:0]	050703f6
done	1
interrupt	0
clk_period	10000 ps

Figure 4.7: First simulated instance of video data from the Rx Subsystem via `video_out_tdata[23:0]`

Figure 4.8 illustrates the same time instance as Figure 4.7, however it is zoomed in for greater signal detail. Note that the packets of data are exactly the same as the first packets of data being sent from VTPG, as discussed and illustrated in Figure 4.3. Which makes sense, since the video data transmitted from the VTPG, converted into CSI2, and then converted back into video data should have the same values. Observe that the Rx video data is slightly delayed compared to the VTPG video data and only transmits in batches of four packets. However, the pixel data remains exactly the same. Essentially proving that information integrity is kept after going through the CSI2 Tx Subsystem.

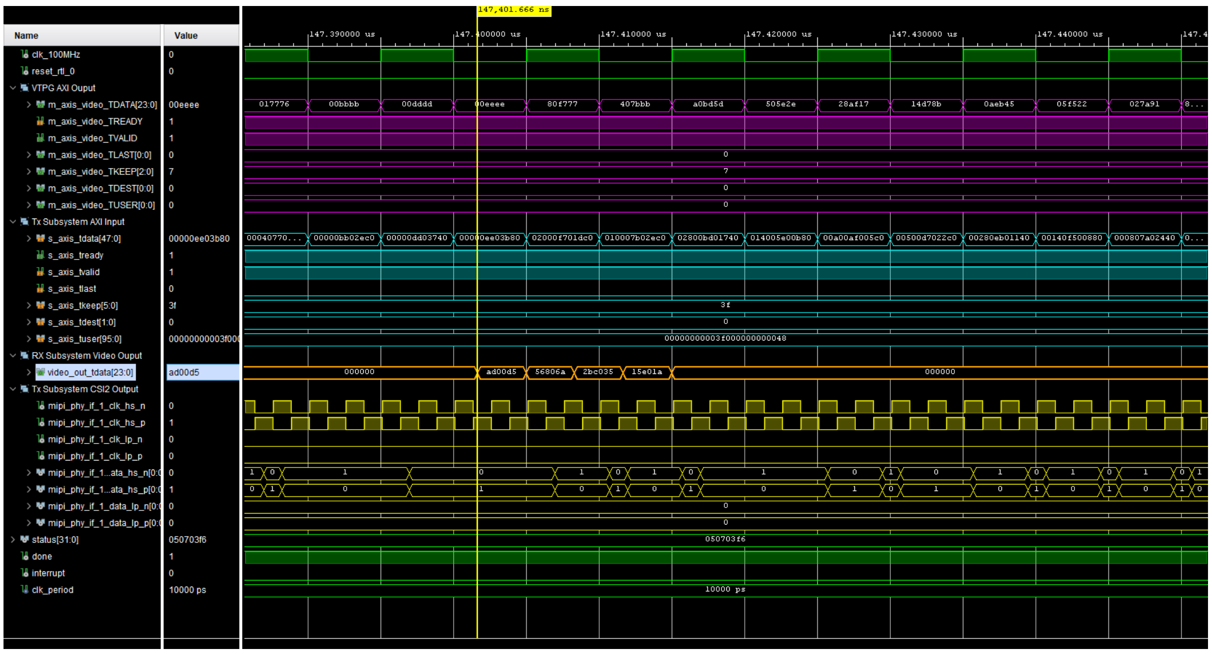


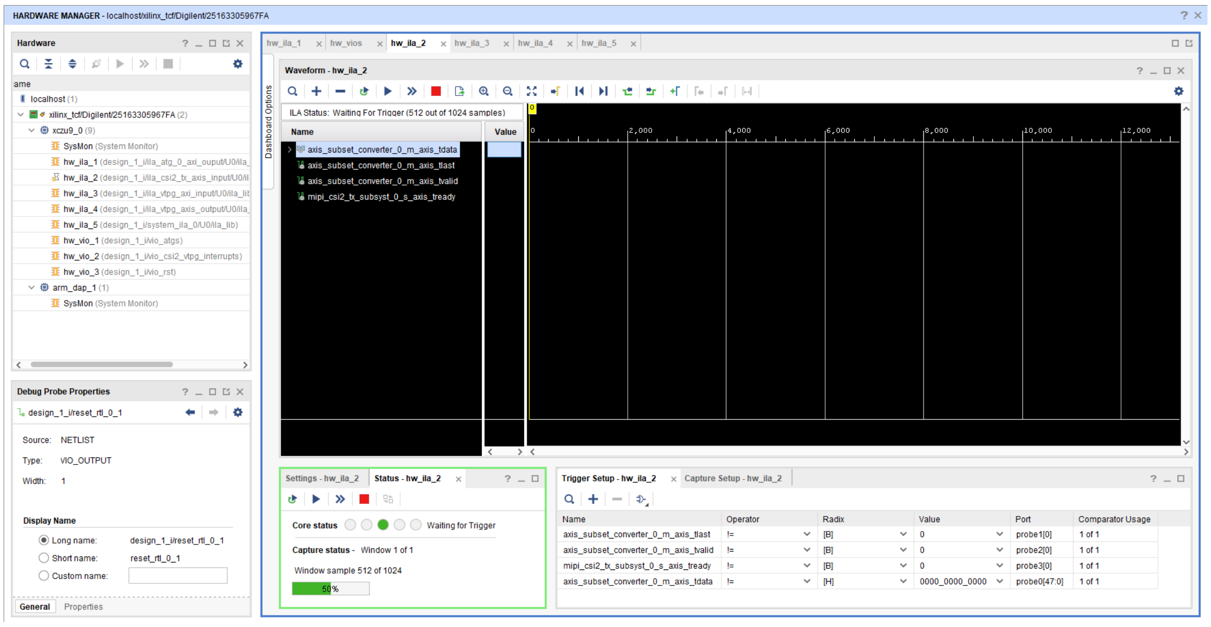
Figure 4.8: Zoomed in Figure 4.7

4.2 Hardware Signal Tap

When implementing a design/program to an embedded system, it is crucial to investigate how the signals behave on the actual hardware. This is where hardware "signal tap" can be useful. The signal tap can be realized in Vivado with ILAs and VIOs, as mentioned in sections 3.7.4 - 3.7.7, via the Hardware Manager. When running a signal tap the ILAs have to be triggered to display the signal waves in the WFW. Throughout this section, the results will be generated by first setting the external reset to ’1’ with the VIO controlling the system reset (which is active high), then starting the ILA and letting it wait for its trigger, then the reset is set to ’0’ which disables the reset and initiates system operations. This should after a short time trigger any ILA with reasonable conditions set for the triggers. This section will provide the resulting signal taps on the implemented designs.

4.2.1 Edited CSI2 Tx Subsystem Example Design with ATG

When using an ILA in the Hardware Manager it is important to specify the desired triggers that will trigger the ILA and display the results. This means that conditions have to be set that tell the ILA to present the current signals in a certain time frame. In Figure 4.9 the ILA monitoring the AXI4 "input" interface to the CSI2 Tx Subsystem is displayed. Though, note that no waveforms are showing in the WFW and the ILA is "Waiting for Trigger". This is due to the fact that none of the conditions for the triggers, as seen below the WFW, are being met. I.e all signals are constantly 0, which means no information is being transferred via these channels. This is a problem, since it effectively means that no test image is being produced from the VTPG.



HARDWARE MANAGER - localhost/xilinx_tcf/Digilent/25163305967FA

Waveform - hw_ila_2

ILA Status: Waiting For Trigger (512 out of 1024 samples)

Name	Value
axis_subset_converter_0_m_axis_tdata	
axis_subset_converter_0_m_axis_tlast	
axis_subset_converter_0_m_axis_tvalid	
mipi_csi2_tx_subsyst_0_s_axis_tready	

Core status: Waiting for Trigger

Capture status - Window 1 of 1

Window sample 512 of 1024

50%

Trigger Setup - hw_ila_2

Name	Operator	Radix	Value	Port	Comparator Usage
axis_subset_converter_0_m_axis_tlast	!=	[B]	0	probe1[0]	1 of 1
axis_subset_converter_0_m_axis_tvalid	!=	[B]	0	probe2[0]	1 of 1
mipi_csi2_tx_subsyst_0_s_axis_tready	!=	[B]	0	probe3[0]	1 of 1
axis_subset_converter_0_m_axis_tdata	!=	[H]	0000_0000_0000	probe0[47:0]	1 of 1

Debug Probe Properties: design_1_i/reset_rtl_0_1

Source: NETLIST

Type: VIO_OUTPUT

Width: 1

Display Name

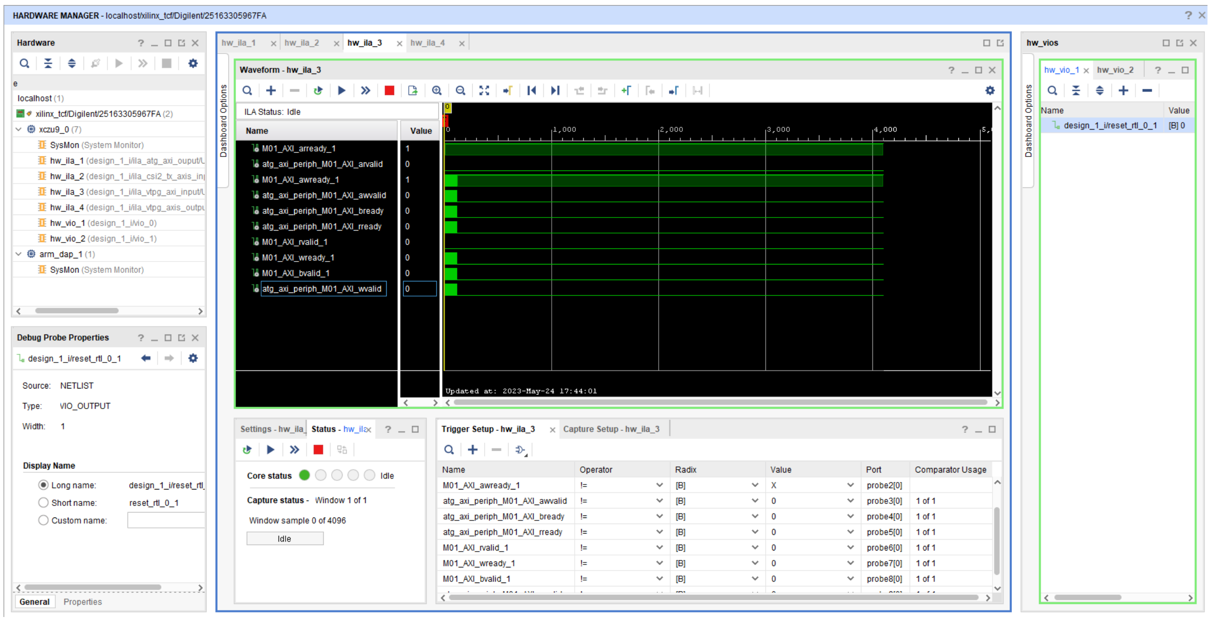
Long name: design_1_i/reset_rtl_0_1

Short name: reset_rtl_0_1

Custom name:

Figure 4.9: Signal tap of AXI4 input interface for CSI2 Tx Subsystem

This problem stems from what is most likely a problem with the initialization of the IPs in the design. This means that the AXI4-Lite control signals (from the ATGs in this case) that are supposed to initialize the IPs (especially VTPG and Tx Subsystem) are not behaving as desired. Refer to Figure 4.10 where the AXI4-Lite input to the VTPG is displayed. Here a quick burst of control signals are portrayed, which after a short time go to 0 or 1 and stays constant. A zoomed in version of Figure 4.10 can be seen in Figure 4.11 . These signals are similar to the AXI4-Lite VTPG input signals displayed in Figure 4.12, which is pulled directly from simulations of the example design. These two figures should correlated exactly for respective AXI4-Lite signals since nothing in the VTPG or the Tx Subsystem has been changed. Though they slightly differ in the amount of peaks each control signal has.



HARDWARE MANAGER - localhost/xilinx_tcf/Digilent/25163305967FA

Waveform - hw_ila_3

ILA Status: Idle

Name	Value
M01_AXI_arready_1	1
atg_axi_periph_M01_AXI_arvalid	0
M01_AXI_awready_1	1
atg_axi_periph_M01_AXI_awvalid	0
atg_axi_periph_M01_AXI_bready	0
atg_axi_periph_M01_AXI_rready	0
M01_AXI_rvalid_1	0
M01_AXI_wready_1	0
M01_AXI_bvalid_1	0
atg_axi_periph_M01_AXI_wvalid	0

Updated at: 2023-May-24 17:44:01

Core status: Idle

Capture status - Window 1 of 1

Window sample 0 of 4096

Idle

Trigger Setup - hw_ila_3

Name	Operator	Radix	Value	Port	Comparator Usage
M01_AXI_awready_1	!=	[B]	X	probe2[0]	
atg_axi_periph_M01_AXI_awvalid	!=	[B]	0	probe3[0]	1 of 1
atg_axi_periph_M01_AXI_bready	!=	[B]	0	probe4[0]	1 of 1
atg_axi_periph_M01_AXI_rready	!=	[B]	0	probe5[0]	1 of 1
M01_AXI_rvalid_1	!=	[B]	0	probe6[0]	1 of 1
M01_AXI_wready_1	!=	[B]	0	probe7[0]	1 of 1
M01_AXI_bvalid_1	!=	[B]	0	probe8[0]	1 of 1

hw_vios

Name	Value
design_1_i/reset_rtl_0_1	[B] 0

Figure 4.10: Signal tap of AXI4-Lite input to the VTPG

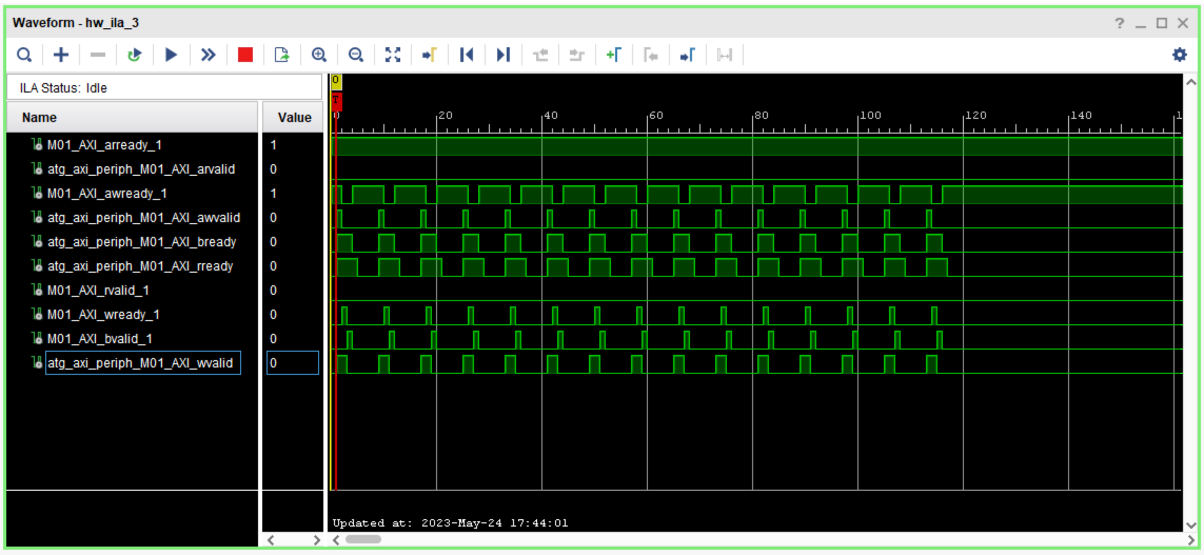

Waveform - hw_ila_3
ILA Status: Idle
Name
Value
M01_AXI_arready_1 1
atg_axi_periph_M01_AXI_arvalid 0
M01_AXI_awready_1 1
atg_axi_periph_M01_AXI_awvalid 0
atg_axi_periph_M01_AXI_bready 0
atg_axi_periph_M01_AXI_rready 0
M01_AXI_rvalid_1 0
M01_AXI_wready_1 0
M01_AXI_bvalid_1 0
atg_axi_periph_M01_AXI_wvalid 0
0
20
40
60
80
100
120
140
Updated at: 2023-May-24 17:44:01

Figure 4.11: Zoomed in Figure 4.10

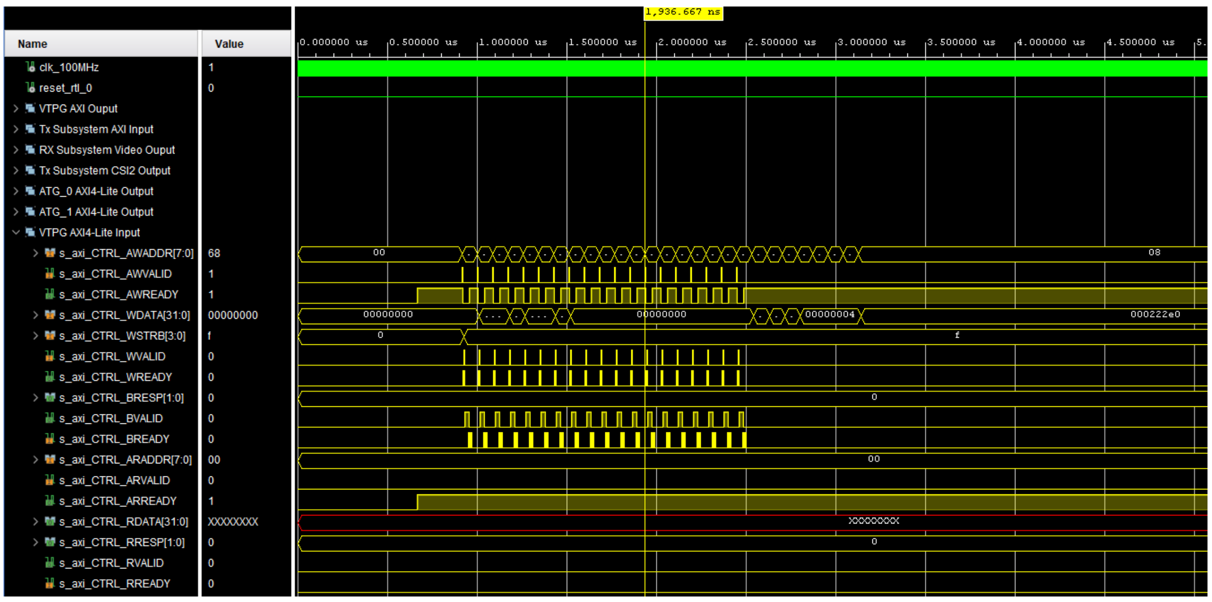
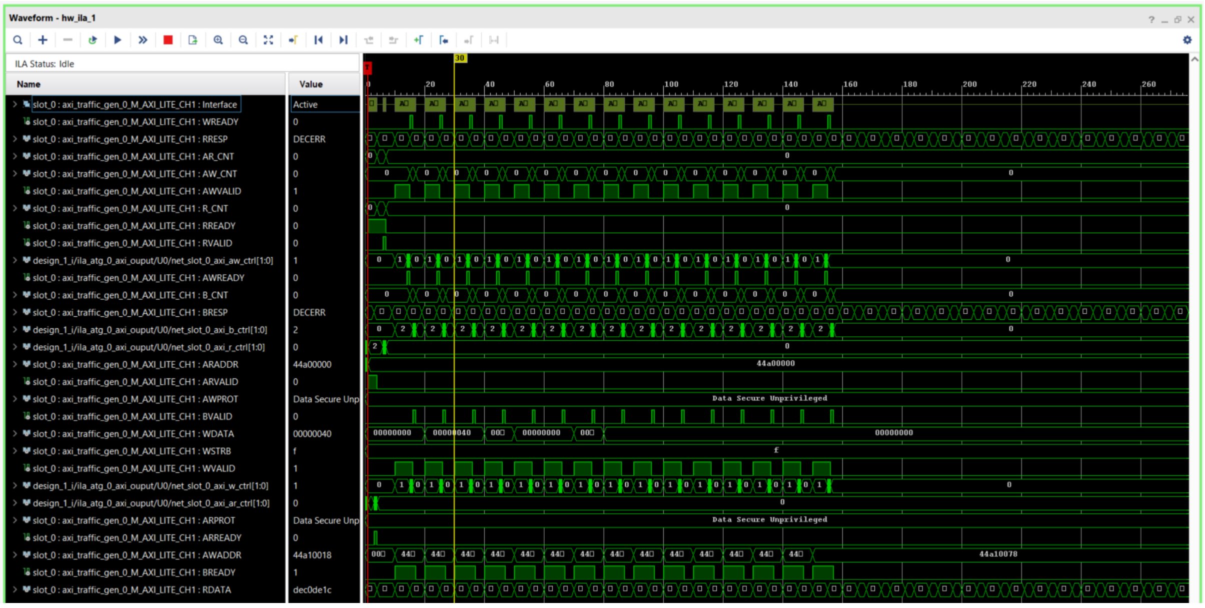


Figure 4.12: Simulated signals of AXI4-Lite input to the VTPG

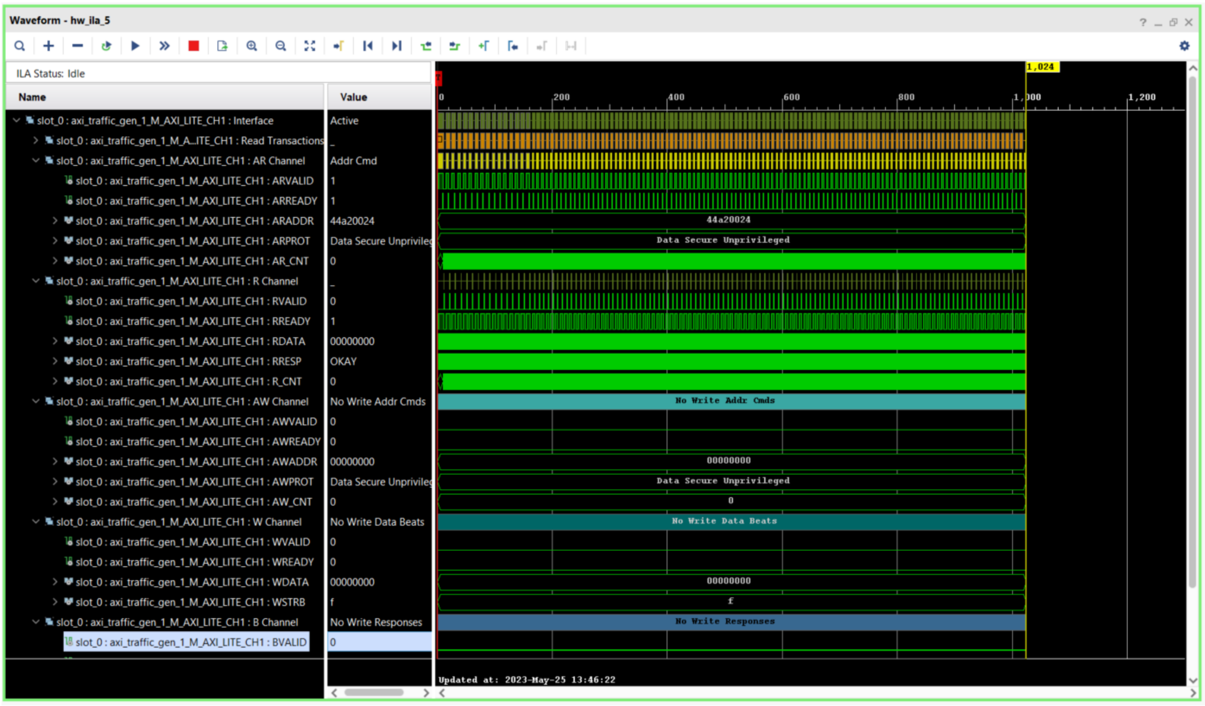
Additionally, refer to Figure 4.13 and Figure 4.14 where the AXI4-Lite outputs from the ATG0 and ATG1, respectively, are displayed. Here it can be observed that the ATGs are successfully sending AXI4-Lite signals on the hardware. Also observe the simulated AXI4-Lite output signals for both ATGs in Figure 4.16, for comparison. The simulated signals for the ATG0 can be argued look similar to the signals in the ATG0 signal tap. Though, while the signals are being transmitted instantly from ATG0 in the simulations (Figure 4.16), there appears to be almost no variation in the signals from ATG1. This is most likely due ATG1 taking longer to start transmitting data after the system starts up.


Waveform - hw_ila_1

ILA Status: Idle

Name	Value
slot_0 : axi_traffic_gen_0_M_AXI_LITE_CH1 : Interface	Active
slot_0 : axi_traffic_gen_0_M_AXI_LITE_CH1 : WREADY	0
slot_0 : axi_traffic_gen_0_M_AXI_LITE_CH1 : RRESP	DECERR
slot_0 : axi_traffic_gen_0_M_AXI_LITE_CH1 : AR_CNT	0
slot_0 : axi_traffic_gen_0_M_AXI_LITE_CH1 : AW_CNT	0
slot_0 : axi_traffic_gen_0_M_AXI_LITE_CH1 : AWVALID	1
slot_0 : axi_traffic_gen_0_M_AXI_LITE_CH1 : R_CNT	0
slot_0 : axi_traffic_gen_0_M_AXI_LITE_CH1 : RREADY	0
slot_0 : axi_traffic_gen_0_M_AXI_LITE_CH1 : RVALID	0
design_1_i/ila_atg_0_axi_ouput/U0/net_slot_0_axi_aw_ctrl[1:0]	1
slot_0 : axi_traffic_gen_0_M_AXI_LITE_CH1 : AWREADY	0
slot_0 : axi_traffic_gen_0_M_AXI_LITE_CH1 : B_CNT	0
slot_0 : axi_traffic_gen_0_M_AXI_LITE_CH1 : BRESP	DECERR
design_1_i/ila_atg_0_axi_ouput/U0/net_slot_0_axi_b_ctrl[1:0]	2
design_1_i/ila_atg_0_axi_ouput/U0/net_slot_0_axi_r_ctrl[1:0]	0
slot_0 : axi_traffic_gen_0_M_AXI_LITE_CH1 : ARADDR	44a00000
slot_0 : axi_traffic_gen_0_M_AXI_LITE_CH1 : ARVALID	0
slot_0 : axi_traffic_gen_0_M_AXI_LITE_CH1 : AWPROT	Data Secure Unp
slot_0 : axi_traffic_gen_0_M_AXI_LITE_CH1 : BVALID	0
slot_0 : axi_traffic_gen_0_M_AXI_LITE_CH1 : WDATA	00000040
slot_0 : axi_traffic_gen_0_M_AXI_LITE_CH1 : WSTRB	f
slot_0 : axi_traffic_gen_0_M_AXI_LITE_CH1 : WVALID	1
design_1_i/ila_atg_0_axi_ouput/U0/net_slot_0_axi_w_ctrl[1:0]	1
design_1_i/ila_atg_0_axi_ouput/U0/net_slot_0_axi_ar_ctrl[1:0]	0
slot_0 : axi_traffic_gen_0_M_AXI_LITE_CH1 : ARPROT	Data Secure Unp
slot_0 : axi_traffic_gen_0_M_AXI_LITE_CH1 : ARREADY	0
slot_0 : axi_traffic_gen_0_M_AXI_LITE_CH1 : AWADDR	44a10018
slot_0 : axi_traffic_gen_0_M_AXI_LITE_CH1 : BREADY	1
slot_0 : axi_traffic_gen_0_M_AXI_LITE_CH1 : RDATA	dec0de1c

Figure 4.13: Signal tap AXI4-Lite output from ATG0

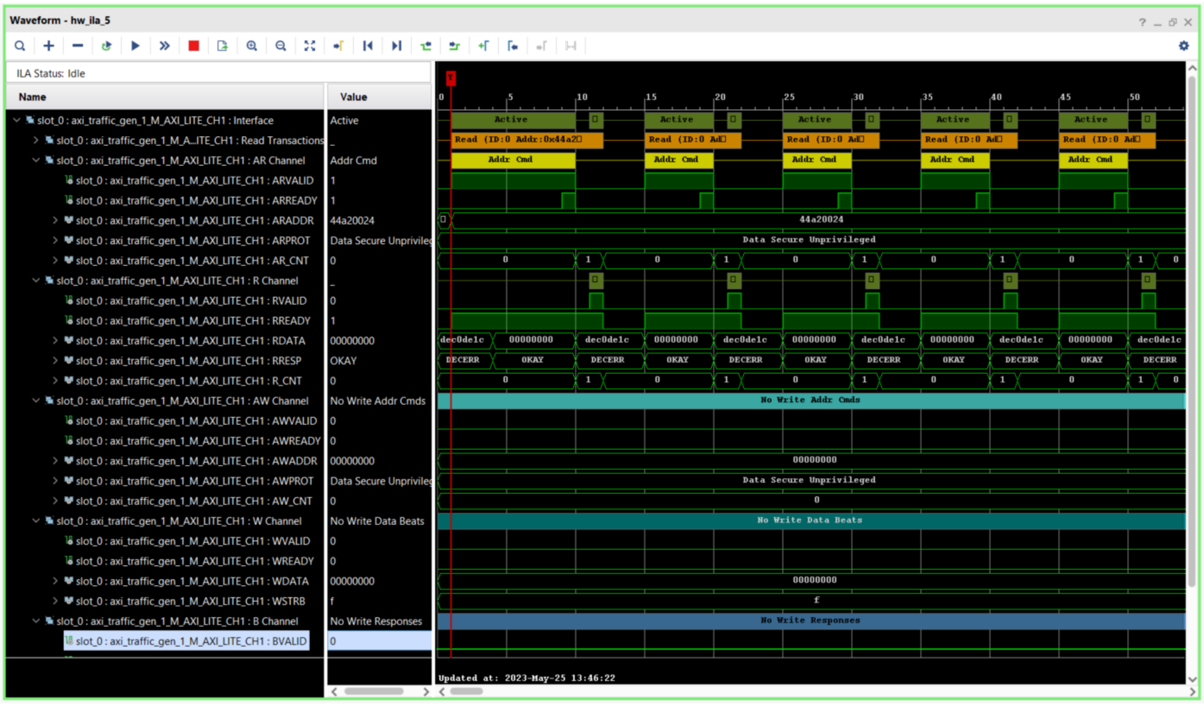

Waveform - hw_ila_5

ILA Status: Idle

Name	Value
slot_0 : axi_traffic_gen_1_M_AXI_LITE_CH1 : Interface	Active
slot_0 : axi_traffic_gen_1_M_A_ITE_CH1 : Read Transactions	_
slot_0 : axi_traffic_gen_1_M_AXI_LITE_CH1 : AR Channel	Addr Cmd
slot_0 : axi_traffic_gen_1_M_AXI_LITE_CH1 : ARVALID	1
slot_0 : axi_traffic_gen_1_M_AXI_LITE_CH1 : ARREADY	1
slot_0 : axi_traffic_gen_1_M_AXI_LITE_CH1 : ARADDR	44a20024
slot_0 : axi_traffic_gen_1_M_AXI_LITE_CH1 : ARPROT	Data Secure Unprivileg
slot_0 : axi_traffic_gen_1_M_AXI_LITE_CH1 : AR_CNT	0
slot_0 : axi_traffic_gen_1_M_AXI_LITE_CH1 : R Channel	_
slot_0 : axi_traffic_gen_1_M_AXI_LITE_CH1 : RVALID	0
slot_0 : axi_traffic_gen_1_M_AXI_LITE_CH1 : RREADY	1
slot_0 : axi_traffic_gen_1_M_AXI_LITE_CH1 : RDATA	00000000
slot_0 : axi_traffic_gen_1_M_AXI_LITE_CH1 : RRESP	OKAY
slot_0 : axi_traffic_gen_1_M_AXI_LITE_CH1 : R_CNT	0
slot_0 : axi_traffic_gen_1_M_AXI_LITE_CH1 : AW Channel	No Write Addr Cmds
slot_0 : axi_traffic_gen_1_M_AXI_LITE_CH1 : AWVALID	0
slot_0 : axi_traffic_gen_1_M_AXI_LITE_CH1 : AWREADY	0
slot_0 : axi_traffic_gen_1_M_AXI_LITE_CH1 : AWADDR	00000000
slot_0 : axi_traffic_gen_1_M_AXI_LITE_CH1 : AWPROT	Data Secure Unprivileg
slot_0 : axi_traffic_gen_1_M_AXI_LITE_CH1 : AW_CNT	0
slot_0 : axi_traffic_gen_1_M_AXI_LITE_CH1 : W Channel	No Write Data Beats
slot_0 : axi_traffic_gen_1_M_AXI_LITE_CH1 : WVALID	0
slot_0 : axi_traffic_gen_1_M_AXI_LITE_CH1 : WREADY	0
slot_0 : axi_traffic_gen_1_M_AXI_LITE_CH1 : WDATA	00000000
slot_0 : axi_traffic_gen_1_M_AXI_LITE_CH1 : WSTRB	f
slot_0 : axi_traffic_gen_1_M_AXI_LITE_CH1 : B Channel	No Write Responses
slot_0 : axi_traffic_gen_1_M_AXI_LITE_CH1 : BVALID	0

Updated at: 2023-May-25 13:46:22

Figure 4.14: Signal tap AXI4-Lite output from ATG1



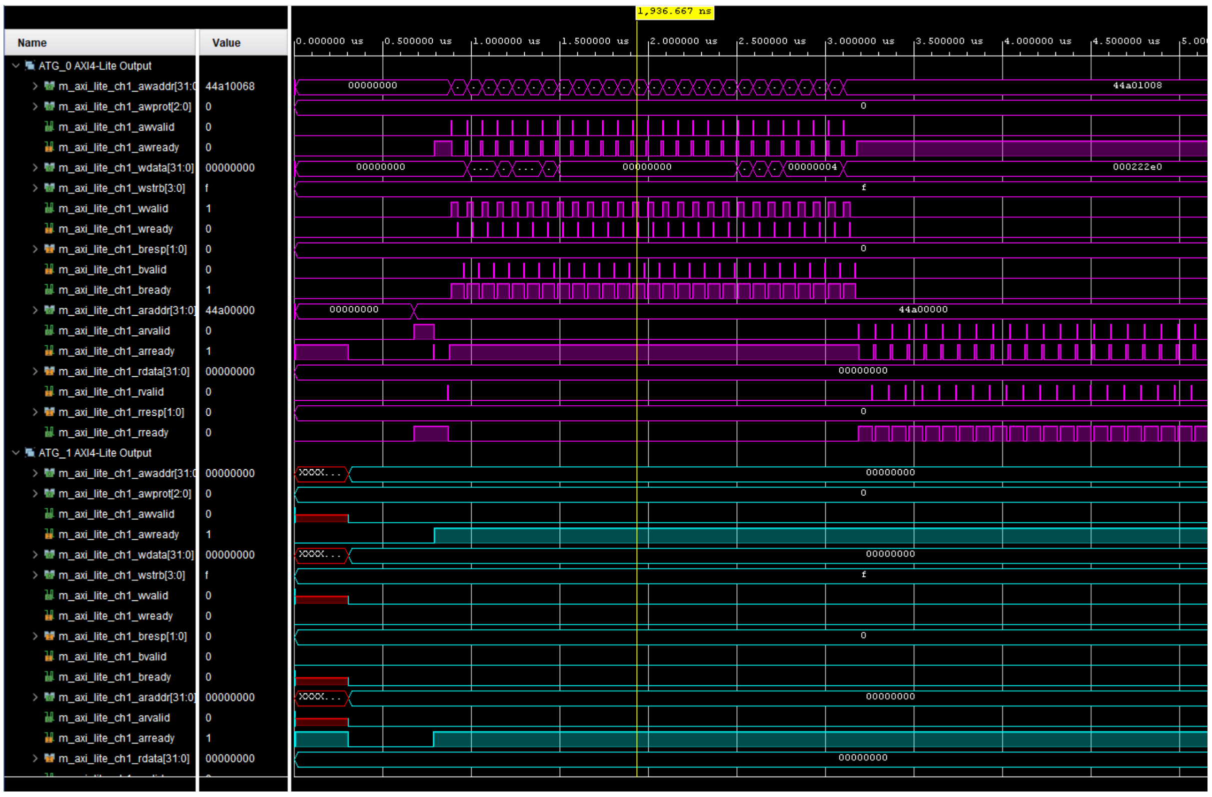
Waveform - hw_ila_5

ILA Status: Idle

Name	Value
slot_0 : axi_traffic_gen_1_M_AXI_LITE_CH1 : Interface	Active
slot_0 : axi_traffic_gen_1_M_A..ITE_CH1 : Read Transactions	_
slot_0 : axi_traffic_gen_1_M_AXI_LITE_CH1 : AR Channel	Addr Cmd
slot_0 : axi_traffic_gen_1_M_AXI_LITE_CH1 : ARVALID	1
slot_0 : axi_traffic_gen_1_M_AXI_LITE_CH1 : ARREADY	1
slot_0 : axi_traffic_gen_1_M_AXI_LITE_CH1 : ARADDR	44a20024
slot_0 : axi_traffic_gen_1_M_AXI_LITE_CH1 : ARPROT	Data Secure Unprivileg
slot_0 : axi_traffic_gen_1_M_AXI_LITE_CH1 : AR_CNT	0
slot_0 : axi_traffic_gen_1_M_AXI_LITE_CH1 : R Channel	_
slot_0 : axi_traffic_gen_1_M_AXI_LITE_CH1 : RVALID	0
slot_0 : axi_traffic_gen_1_M_AXI_LITE_CH1 : RREADY	1
slot_0 : axi_traffic_gen_1_M_AXI_LITE_CH1 : RDATA	00000000
slot_0 : axi_traffic_gen_1_M_AXI_LITE_CH1 : RRESP	OKAY
slot_0 : axi_traffic_gen_1_M_AXI_LITE_CH1 : R_CNT	0
slot_0 : axi_traffic_gen_1_M_AXI_LITE_CH1 : AW Channel	No Write Addr Cmds
slot_0 : axi_traffic_gen_1_M_AXI_LITE_CH1 : AWVALID	0
slot_0 : axi_traffic_gen_1_M_AXI_LITE_CH1 : AWREADY	0
slot_0 : axi_traffic_gen_1_M_AXI_LITE_CH1 : AWADDR	00000000
slot_0 : axi_traffic_gen_1_M_AXI_LITE_CH1 : AWPROT	Data Secure Unprivileg
slot_0 : axi_traffic_gen_1_M_AXI_LITE_CH1 : AW_CNT	0
slot_0 : axi_traffic_gen_1_M_AXI_LITE_CH1 : W Channel	No Write Data Beats
slot_0 : axi_traffic_gen_1_M_AXI_LITE_CH1 : WVALID	0
slot_0 : axi_traffic_gen_1_M_AXI_LITE_CH1 : WREADY	0
slot_0 : axi_traffic_gen_1_M_AXI_LITE_CH1 : WDATA	00000000
slot_0 : axi_traffic_gen_1_M_AXI_LITE_CH1 : WSTRB	f
slot_0 : axi_traffic_gen_1_M_AXI_LITE_CH1 : B Channel	No Write Responses
slot_0 : axi_traffic_gen_1_M_AXI_LITE_CH1 : BVALID	0

Updated at: 2023-May-25 13:46:22

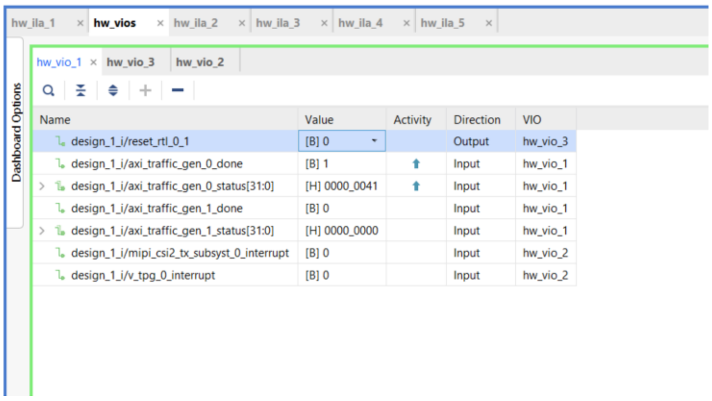
Figure 4.15: Zoomed in Figure 4.14



Name	Value
ATG_0 AXI4-Lite Output	
m_axi_lite_ch1_awaddr[31:0]	44a10068
m_axi_lite_ch1_awprot[2:0]	0
m_axi_lite_ch1_awvalid	0
m_axi_lite_ch1_awready	0
m_axi_lite_ch1_wdata[31:0]	00000000
m_axi_lite_ch1_wstrb[3:0]	f
m_axi_lite_ch1_wvalid	1
m_axi_lite_ch1_wready	0
m_axi_lite_ch1_bresp[1:0]	0
m_axi_lite_ch1_bvalid	0
m_axi_lite_ch1_bready	1
m_axi_lite_ch1_araddr[31:0]	44a00000
m_axi_lite_ch1_arvalid	0
m_axi_lite_ch1_arready	1
m_axi_lite_ch1_rdata[31:0]	00000000
m_axi_lite_ch1_rvalid	0
m_axi_lite_ch1_rresp[1:0]	0
m_axi_lite_ch1_rready	0
ATG_1 AXI4-Lite Output	
m_axi_lite_ch1_awaddr[31:0]	00000000
m_axi_lite_ch1_awprot[2:0]	0
m_axi_lite_ch1_awvalid	0
m_axi_lite_ch1_awready	1
m_axi_lite_ch1_wdata[31:0]	00000000
m_axi_lite_ch1_wstrb[3:0]	f
m_axi_lite_ch1_wvalid	0
m_axi_lite_ch1_wready	0
m_axi_lite_ch1_bresp[1:0]	0
m_axi_lite_ch1_bvalid	0
m_axi_lite_ch1_bready	0
m_axi_lite_ch1_araddr[31:0]	00000000
m_axi_lite_ch1_arvalid	0
m_axi_lite_ch1_arready	1
m_axi_lite_ch1_rdata[31:0]	00000000

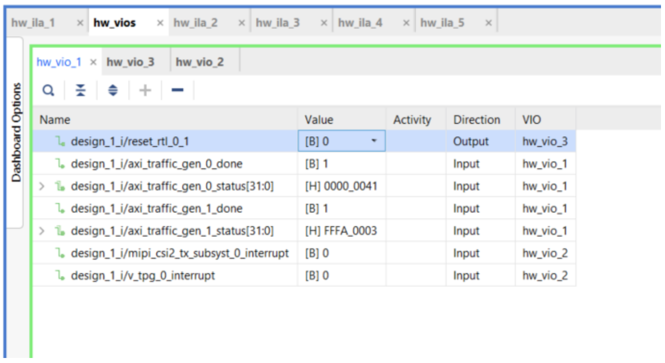
Figure 4.16: Simulated signals of AXI4-Lite ouput from both ATG0 and ATG1

The aforementioned delay present in ATG1 was also observed live, while switching the vio reset on-and-off in the Hardware Manager. Refer to Figure 4.17 where the VIO ports of all VIOs in the system are present, including the system reset and status ouputs from the two ATGs. In this time instance that the figure shows, the system reset signal was set from '1' (active) to '0' (non-active). The `status` and `done` signals for ATG0 changed almost instantly after deactivating the reset, while the `status` and `done` signals for ATG1 remained unchanged. Now observe Figure 4.18 which is the results of the reset signal being kept at 0 until ATG1's signals change. This took place after roughly 10 real-time seconds. Essentially providing a viable explanation to why no changes were observed in the simulations in Figure 4.16, since these only span in the scale of µ*s*. Longer simulations would be extremely time and memory demanding.



Name	Value	Activity	Direction	VIO
design_1_i/reset_rtl_0_1	[B] 0		Output	hw_vio_3
design_1_i/axi_traffic_gen_0_done	[B] 1	⬆	Input	hw_vio_1
design_1_i/axi_traffic_gen_0_status[31:0]	[H] 0000_0041	⬆	Input	hw_vio_1
design_1_i/axi_traffic_gen_1_done	[B] 0		Input	hw_vio_1
design_1_i/axi_traffic_gen_1_status[31:0]	[H] 0000_0000		Input	hw_vio_1
design_1_i/mipi_csi2_tx_subsyst_0_interrupt	[B] 0		Input	hw_vio_2
design_1_i/v_tpg_0_interrupt	[B] 0		Input	hw_vio_2

Figure 4.17: VIO reset signal set from 1 to 0



Name	Value	Activity	Direction	VIO
design_1_i/reset_rtl_0_1	[B] 0		Output	hw_vio_3
design_1_i/axi_traffic_gen_0_done	[B] 1		Input	hw_vio_1
design_1_i/axi_traffic_gen_0_status[31:0]	[H] 0000_0041		Input	hw_vio_1
design_1_i/axi_traffic_gen_1_done	[B] 1		Input	hw_vio_1
design_1_i/axi_traffic_gen_1_status[31:0]	[H] FFFA_0003		Input	hw_vio_1
design_1_i/mipi_csi2_tx_subsyst_0_interrupt	[B] 0		Input	hw_vio_2
design_1_i/v_tpg_0_interrupt	[B] 0		Input	hw_vio_2

Figure 4.18: VIO reset signal remaining a 0 for some time

4.3 General Discussion

Even though no concrete proof has been found explaining why the VTPG is not producing any video data, qualified guesses can be made on the results presented. In the hardware, there are AXI4-Lite signals being transmitted from both ATGs and there are AXI4-Lite signals being accepted by the VTPG. Though, nothing is coming out on the other end of the VTPG. An assumption can be made that the cause for this is wrongful AXI4-Lite data, rather than the absence of AXI4-Lite data. To fix this issue, more time is needed to delve deeper into what exactly is being sent on the hardware and to try different approaches on how to generate the AXI4-Lite signals in a controllable and understandable way.

One of these options is to completely abandon the idea of using AXI Traffic Generators as a way to generate the AXI4-Lite signals, and instead focus on generating the control signals with C code drivers for the PS. Another option is to explore other ways of using the ATGs, apart from the current "Custom Mode" implemented. The issue could, reasonably, spur from the fact that the AXI4-Lite signals are being produced based on COE files, and that these files are exclusively meant for the unedited example design.

5 Conclusion and Future Work

In this degree project, research and development of a simple solution has been conducted for the output stage of an Interface Unit, as a proof of concept that image injection can be achieved with the currently available hardware and software. A preliminary design for the output stage has been implemented on the hardware (MPSoC) that generates signals in some parts of the design. The signals in the hardware have, in turn, been compared to signals generated in a simulation to examine the parts of the design that are failing. Based on these comparisons, this is believed to be an issue with initialization of these parts. No concrete final fully implemented solution has been found.

One of the main reasons for not achieving a full implementation of a solution was the issue with wrong hardware. The part that this thesis report mainly focused on, i.e the output stage, was heavily based on the notion of getting a certain hardware (TEF0010) to handle the transmitting of GMSL2. However, after some weeks of waiting for the card to arrive and then some weeks of trying to get the card to work, it was found that the card was a fully custom card designed by the current provider of the IU solution. Thus, since the card is fully custom, the associated schematics and technical reference manuals are fully confidential. Without these, no implementation can realistically be achieved on the current hardware. The only real option to solve this issue is to order the card that was originally intended, since schematics are available for this card, or any other similar card with available schematics and/or technical reference manuals.

Another reason to why the implementation was not achieved within this thesis was the pure complexity and size of the project. Even though the full project was divided into three parts, the original scope was found to have unreasonably high complexity and broad goals. However, even after limiting the scope of the thesis it still proved to be too complex to fully implement even a simple design. This would require far more time and resources to effectively complete. In fact, a meeting was had towards the end of the project with the company currently providing the IU solution. They explained how their final solution had taken roughly five years, with multiple experienced engineers working on it, to complete. Providing some indication to the size of the project.

Despite this, the work presented in this thesis can be used as a foundation for future work within the area. This thesis has proven that it is possible to edit and customize the hardware, and given enough time I personally believe that a simple implementation could be achieved. This simple solution could then be worked further upon to enable more control, higher data throughput, more advanced functions and higher automation. Though, the first step to this progression is to either obtain sufficient information regarding the current hardware, or obtain alternative hardware that already has sufficient technical information available to the public. Additionally, all the hardware used is based on the already aforementioned available IU solution. However, there are other commercial solutions that could offer greater support for customization straight out of the box. This is another venue that could be explored in greater detail for future work within this area.

Appendices

A.1 Block Diagrams

A.1.1 CSI2 Tx Subsystem Example Design

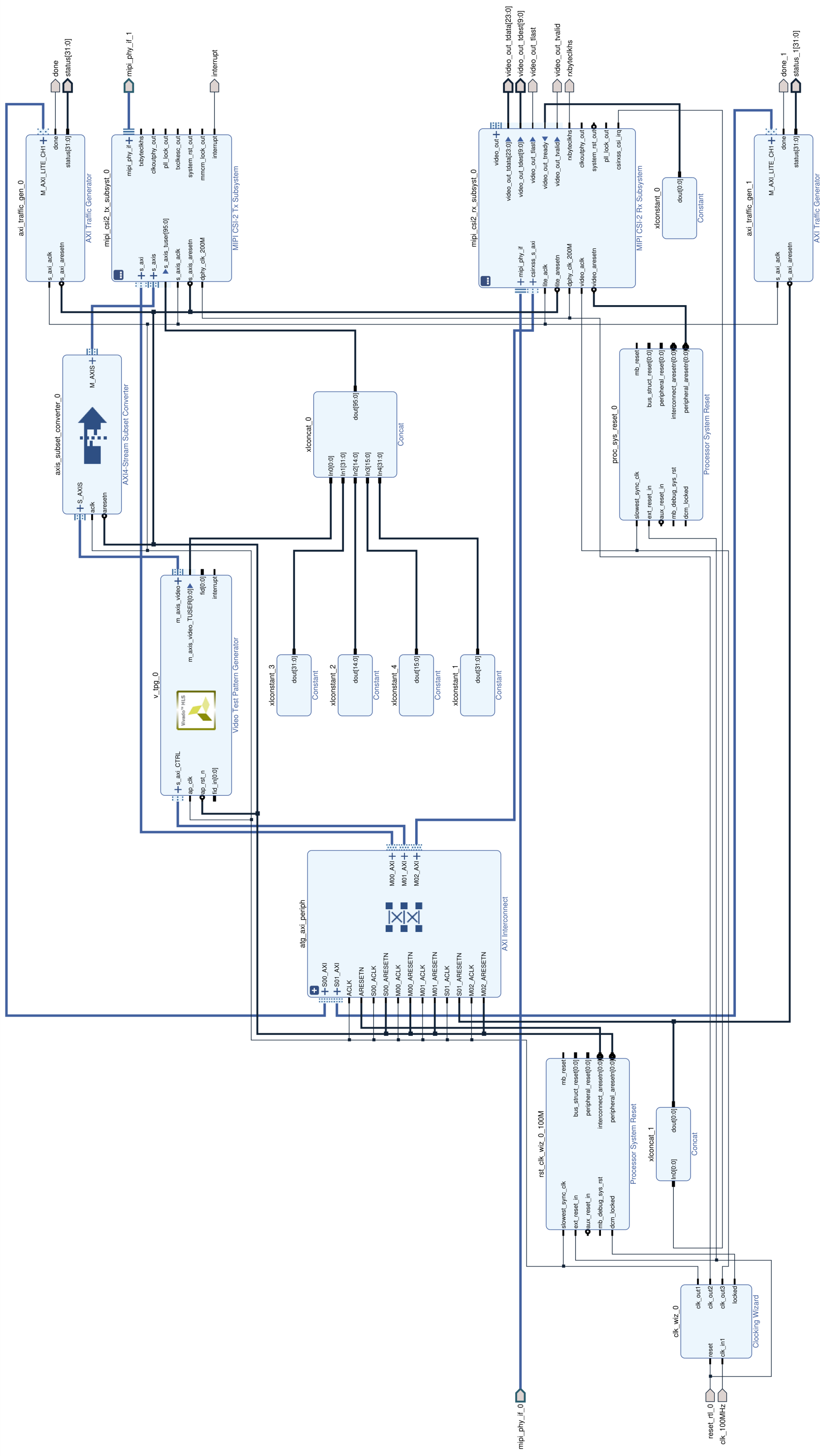


Figure A.1.1: CSI2 Tx Subsystem Example Design

A.1.2 Edited CSI2 Tx Subsystem Example Design with ATG

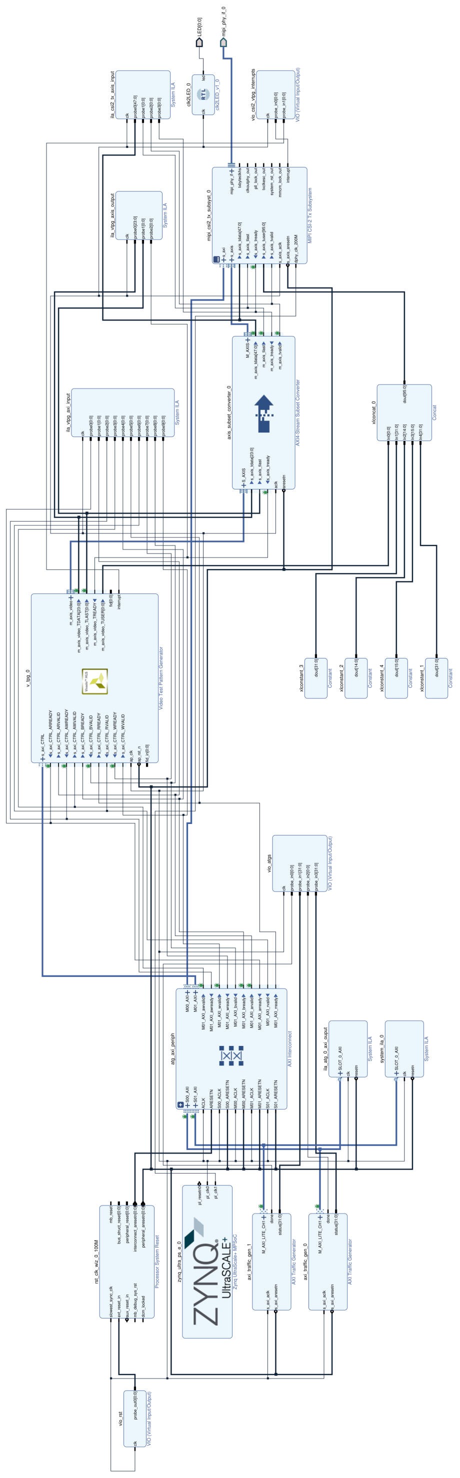


Figure A.1.2: Edited CSI2 Tx Subsystem Example Design with ATG

A.1.3 Edited CSI2 Tx Subsystem Example Design without ATG

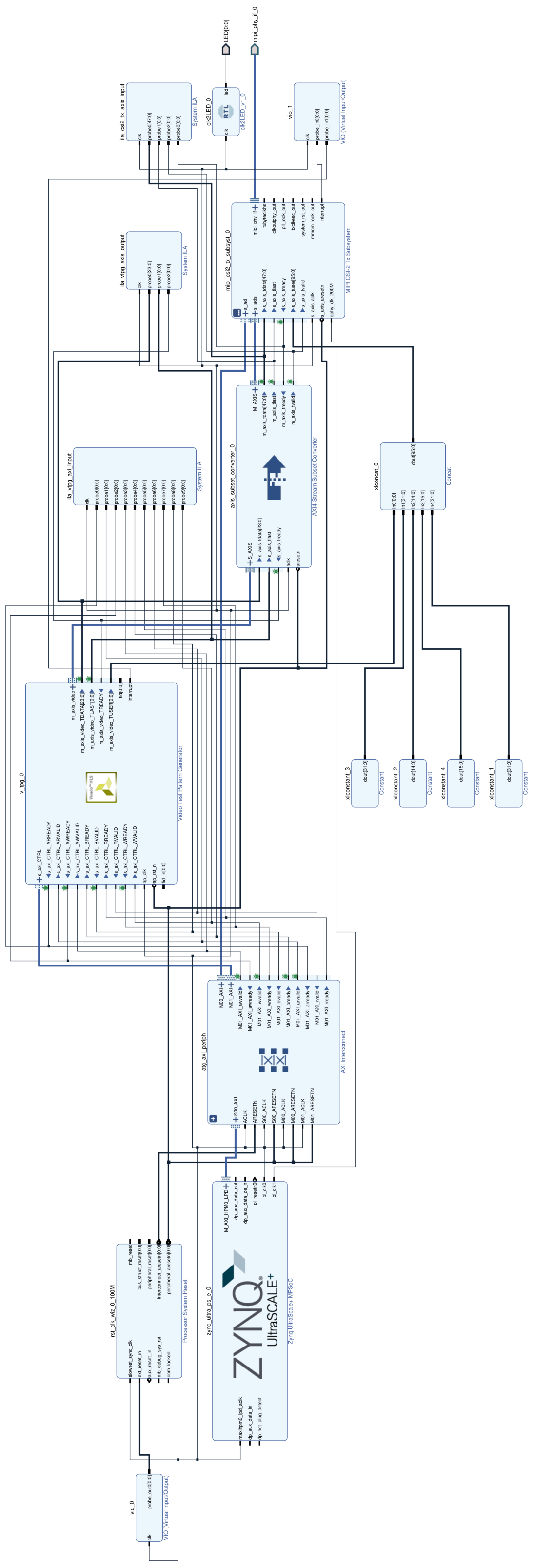


Figure A.1.3: Edited CSI2 Tx Subsystem Example Design

A.2 VHDL Code and Wrappers

A.2.1 Clock Divider for Blinking LED

```vhdl
----------------------------------------------------------------------------------
-- Company: Volvo Cars
-- Engineer: Anton Lind
--
-- Design Date: 2023-05
-- Design Name: clk2LED.vhd
--
-- Purpose: Clock divider that divides an input clock with 100 000 000
--          and outputs the divided clock.
--          Can be connected to a LED to identify whether the input
--          clock is running properly on the hardware.
----------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity clk2LED is

  Port (
  clk : in std_logic;
  led : out std_logic
  );
end clk2LED;

architecture Behavioral of clk2LED is
    constant divider_c : integer := 100000000;

    signal led_s : std_logic := '0';
    signal cnt_s : integer range 0 to divider_c :=0;

begin
    led <= led_s;

    process(clk)
    begin
        if rising_edge(clk) then
            cnt_s <= cnt_s+1;

            if (cnt_s = divider_c) then
                cnt_s <=0;
                led_s <= not led_s;
            end if;
         end if;
    end process;

end Behavioral;
```

A.2.2 CSI2 Tx Example Design Custom Testbench (VHDL)

```vhdl
----------------------------------------------------------------------------------
-- Company: Volvo Cars
-- Engineer: Anton Lind
--
-- Design Date: 2023-05
-- Design Name: CSI2_Tx_Subsystem_Example_Design_TB
--
-- Purpose: VHDL Testbench for the RGB888 CSI2 Tx Subsystem Example Design.
--          Generates a 100MHz input clock and initializes the input reset signal.
--          Additonally, handles the feedback connection between CSI2 Tx output
--          and CSI2 RX input.
--          A process for generating a simple reset signal for testing can also
--          be found (initially commented out).


----------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity my_tb_1 is
--  Port ( );
end my_tb_1;


architecture Behavioral of my_tb_1 is

component design_1_wrapper
port(
    clk_100MHz : in STD_LOGIC;
    reset_rtl_0 : in STD_LOGIC;

    mipi_phy_if_0_clk_hs_n : in STD_LOGIC;
    mipi_phy_if_0_clk_hs_p : in STD_LOGIC;
    mipi_phy_if_0_clk_lp_n : in STD_LOGIC;
    mipi_phy_if_0_clk_lp_p : in STD_LOGIC;
    mipi_phy_if_0_data_hs_n : in STD_LOGIC_VECTOR ( 0 to 0 );
    mipi_phy_if_0_data_hs_p : in STD_LOGIC_VECTOR ( 0 to 0 );
    mipi_phy_if_0_data_lp_n : in STD_LOGIC_VECTOR ( 0 to 0 );
    mipi_phy_if_0_data_lp_p : in STD_LOGIC_VECTOR ( 0 to 0 );

    done : out STD_LOGIC;
    done_1 : out STD_LOGIC;
    interrupt : out STD_LOGIC;
    mipi_phy_if_1_clk_hs_n : out STD_LOGIC;
    mipi_phy_if_1_clk_hs_p : out STD_LOGIC;
    mipi_phy_if_1_clk_lp_n : out STD_LOGIC;
    mipi_phy_if_1_clk_lp_p : out STD_LOGIC;
    mipi_phy_if_1_data_hs_n : out STD_LOGIC_VECTOR ( 0 to 0 );
    mipi_phy_if_1_data_hs_p : out STD_LOGIC_VECTOR ( 0 to 0 );
    mipi_phy_if_1_data_lp_n : out STD_LOGIC_VECTOR ( 0 to 0 );
    mipi_phy_if_1_data_lp_p : out STD_LOGIC_VECTOR ( 0 to 0 );
    rxbyteclkhs : out STD_LOGIC;
    status : out STD_LOGIC_VECTOR ( 31 downto 0 );
    status_1 : out STD_LOGIC_VECTOR ( 31 downto 0 );
    video_out_tdata : out STD_LOGIC_VECTOR ( 23 downto 0 );
    video_out_tdest : out STD_LOGIC_VECTOR ( 9 downto 0 );
    video_out_tlast : out STD_LOGIC;
    video_out_tvalid : out STD_LOGIC
    );
end component;

--Inputs-----------------------------------------------------------
signal clk_100MHz : STD_LOGIC;
signal reset_rtl_0 : STD_LOGIC := '0';

signal mipi_phy_if_0_clk_hs_n : STD_LOGIC;
signal mipi_phy_if_0_clk_hs_p : STD_LOGIC;
signal mipi_phy_if_0_clk_lp_n : STD_LOGIC;
signal mipi_phy_if_0_clk_lp_p : STD_LOGIC;
signal mipi_phy_if_0_data_hs_n : STD_LOGIC_VECTOR ( 0 to 0 );
signal mipi_phy_if_0_data_hs_p : STD_LOGIC_VECTOR ( 0 to 0 );
signal mipi_phy_if_0_data_lp_n : STD_LOGIC_VECTOR ( 0 to 0 );
signal mipi_phy_if_0_data_lp_p : STD_LOGIC_VECTOR ( 0 to 0 );
-----------------------------------------------------------------

-- Outputs-------------------------------------------------------
signal done :  STD_LOGIC;
signal done_1 :  STD_LOGIC;
signal interrupt :  STD_LOGIC;
signal mipi_phy_if_1_clk_hs_n : STD_LOGIC;
signal mipi_phy_if_1_clk_hs_p : STD_LOGIC;
signal mipi_phy_if_1_clk_lp_n : STD_LOGIC;
signal mipi_phy_if_1_clk_lp_p : STD_LOGIC;
signal mipi_phy_if_1_data_hs_n : STD_LOGIC_VECTOR ( 0 to 0 );
signal mipi_phy_if_1_data_hs_p : STD_LOGIC_VECTOR ( 0 to 0 );
signal mipi_phy_if_1_data_lp_n : STD_LOGIC_VECTOR ( 0 to 0 );
signal mipi_phy_if_1_data_lp_p : STD_LOGIC_VECTOR ( 0 to 0 );
signal rxbyteclkhs :  STD_LOGIC;
signal status :  STD_LOGIC_VECTOR ( 31 downto 0 );
signal status_1 :  STD_LOGIC_VECTOR ( 31 downto 0 );
signal video_out_tdata :  STD_LOGIC_VECTOR ( 23 downto 0 );
signal video_out_tdest :  STD_LOGIC_VECTOR ( 9 downto 0 );
signal video_out_tlast :  STD_LOGIC;
signal video_out_tvalid :  STD_LOGIC;
-----------------------------------------------------------------

-- Defining clock period for a 100MHz clock
constant clk_period : time := 10 ns;


begin
-- Initiate the Unit Under Test (UUT)
uut: design_1_wrapper
port map(
      clk_100MHz => clk_100MHz,
      done => done,
      done_1 => done_1,
      interrupt => interrupt,
      mipi_phy_if_0_clk_hs_n => mipi_phy_if_0_clk_hs_n,
      mipi_phy_if_0_clk_hs_p => mipi_phy_if_0_clk_hs_p,
      mipi_phy_if_0_clk_lp_n => mipi_phy_if_0_clk_lp_n,
      mipi_phy_if_0_clk_lp_p => mipi_phy_if_0_clk_lp_p,
      mipi_phy_if_0_data_hs_n(0) => mipi_phy_if_0_data_hs_n(0),
      mipi_phy_if_0_data_hs_p(0) => mipi_phy_if_0_data_hs_p(0),
      mipi_phy_if_0_data_lp_n(0) => mipi_phy_if_0_data_lp_n(0),
      mipi_phy_if_0_data_lp_p(0) => mipi_phy_if_0_data_lp_p(0),
      mipi_phy_if_1_clk_hs_n => mipi_phy_if_1_clk_hs_n,
      mipi_phy_if_1_clk_hs_p => mipi_phy_if_1_clk_hs_p,
      mipi_phy_if_1_clk_lp_n => mipi_phy_if_1_clk_lp_n,
      mipi_phy_if_1_clk_lp_p => mipi_phy_if_1_clk_lp_p,
      mipi_phy_if_1_data_hs_n(0) => mipi_phy_if_1_data_hs_n(0),
      mipi_phy_if_1_data_hs_p(0) => mipi_phy_if_1_data_hs_p(0),
      mipi_phy_if_1_data_lp_n(0) => mipi_phy_if_1_data_lp_n(0),
      mipi_phy_if_1_data_lp_p(0) => mipi_phy_if_1_data_lp_p(0),
      reset_rtl_0 => reset_rtl_0,
      rxbyteclkhs => rxbyteclkhs,
      status(31 downto 0) => status(31 downto 0),
      status_1(31 downto 0) => status_1(31 downto 0),
      video_out_tdata(23 downto 0) => video_out_tdata(23 downto 0),
      video_out_tdest(9 downto 0) => video_out_tdest(9 downto 0),
      video_out_tlast => video_out_tlast,
      video_out_tvalid => video_out_tvalid
    );

-- Feeding CSI2 outputs of the Tx core to the Rx core as inputs
 mipi_phy_if_0_clk_hs_n <= mipi_phy_if_1_clk_hs_n;
 mipi_phy_if_0_clk_hs_p <= mipi_phy_if_1_clk_hs_p;
 mipi_phy_if_0_clk_lp_n <= mipi_phy_if_1_clk_lp_n;
 mipi_phy_if_0_clk_lp_p <= mipi_phy_if_1_clk_lp_p;
 mipi_phy_if_0_data_hs_n <=  mipi_phy_if_1_data_hs_n;
 mipi_phy_if_0_data_hs_p <=  mipi_phy_if_1_data_hs_p;
 mipi_phy_if_0_data_lp_n <=  mipi_phy_if_1_data_lp_n;
 mipi_phy_if_0_data_lp_p <=  mipi_phy_if_1_data_lp_p;


-- Generating 100MHz clock signals
clk_process : process
begin
    clk_100MHz <= '0';
    wait for clk_period/2;
    clk_100MHz <= '1';
    wait for clk_period/2;
end process;

-- Uncomment the following to generate a test-reset signal
---- Generating rst signal
--rst_process : process
--begin
--    wait for 700 ns;
--    reset_rtl_0 <= '1';
--    wait for 100 ns;
--    reset_rtl_0 <= '0';
--    wait;
--end process;
```

```
end Behavioral;
```

A.2.3 CSI2 Tx Subsystem Example Design VHDL Wrapper

```
--Copyright 1986-2022 Xilinx, Inc. All Rights Reserved.
----------------------------------------------------------------------------------
--Tool Version: Vivado v.2022.2 (win64) Build 3671981 Fri Oct 14 05:00:03 MDT 2022
--Date        : Wed Mar  1 13:08:53 2023
--Host        : 5CG8505X0V running 64-bit major release  (build 9200)
--Command     : generate_target design_1_wrapper.bd
--Design      : design_1_wrapper
--Purpose     : IP block netlist
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
library UNISIM;
use UNISIM.VCOMPONENTS.ALL;
entity design_1_wrapper is
  port (
    clk_100MHz : in STD_LOGIC;
    done : out STD_LOGIC;
    done_1 : out STD_LOGIC;
    interrupt : out STD_LOGIC;
    mipi_phy_if_0_clk_hs_n : in STD_LOGIC;
    mipi_phy_if_0_clk_hs_p : in STD_LOGIC;
    mipi_phy_if_0_clk_lp_n : in STD_LOGIC;
    mipi_phy_if_0_clk_lp_p : in STD_LOGIC;
    mipi_phy_if_0_data_hs_n : in STD_LOGIC_VECTOR ( 0 to 0 );
    mipi_phy_if_0_data_hs_p : in STD_LOGIC_VECTOR ( 0 to 0 );
    mipi_phy_if_0_data_lp_n : in STD_LOGIC_VECTOR ( 0 to 0 );
    mipi_phy_if_0_data_lp_p : in STD_LOGIC_VECTOR ( 0 to 0 );
    mipi_phy_if_1_clk_hs_n : out STD_LOGIC;
    mipi_phy_if_1_clk_hs_p : out STD_LOGIC;
    mipi_phy_if_1_clk_lp_n : out STD_LOGIC;
    mipi_phy_if_1_clk_lp_p : out STD_LOGIC;
    mipi_phy_if_1_data_hs_n : out STD_LOGIC_VECTOR ( 0 to 0 );
    mipi_phy_if_1_data_hs_p : out STD_LOGIC_VECTOR ( 0 to 0 );
    mipi_phy_if_1_data_lp_n : out STD_LOGIC_VECTOR ( 0 to 0 );
    mipi_phy_if_1_data_lp_p : out STD_LOGIC_VECTOR ( 0 to 0 );
    reset_rtl_0 : in STD_LOGIC;
    rxbyteclkhs : out STD_LOGIC;
    status : out STD_LOGIC_VECTOR ( 31 downto 0 );
    status_1 : out STD_LOGIC_VECTOR ( 31 downto 0 );
    video_out_tdata : out STD_LOGIC_VECTOR ( 23 downto 0 );
    video_out_tdest : out STD_LOGIC_VECTOR ( 9 downto 0 );
    video_out_tlast : out STD_LOGIC;
    video_out_tvalid : out STD_LOGIC
  );
end design_1_wrapper;

architecture STRUCTURE of design_1_wrapper is
  component design_1 is
  port (
    mipi_phy_if_0_clk_hs_n : in STD_LOGIC;
    mipi_phy_if_0_clk_hs_p : in STD_LOGIC;
    mipi_phy_if_0_clk_lp_n : in STD_LOGIC;
    mipi_phy_if_0_clk_lp_p : in STD_LOGIC;
    mipi_phy_if_0_data_hs_n : in STD_LOGIC_VECTOR ( 0 to 0 );
    mipi_phy_if_0_data_hs_p : in STD_LOGIC_VECTOR ( 0 to 0 );
    mipi_phy_if_0_data_lp_n : in STD_LOGIC_VECTOR ( 0 to 0 );
    mipi_phy_if_0_data_lp_p : in STD_LOGIC_VECTOR ( 0 to 0 );
    mipi_phy_if_1_clk_hs_n : out STD_LOGIC;
    mipi_phy_if_1_clk_hs_p : out STD_LOGIC;
    mipi_phy_if_1_clk_lp_n : out STD_LOGIC;
    mipi_phy_if_1_clk_lp_p : out STD_LOGIC;
    mipi_phy_if_1_data_hs_n : out STD_LOGIC_VECTOR ( 0 to 0 );
    mipi_phy_if_1_data_hs_p : out STD_LOGIC_VECTOR ( 0 to 0 );
    mipi_phy_if_1_data_lp_n : out STD_LOGIC_VECTOR ( 0 to 0 );
    mipi_phy_if_1_data_lp_p : out STD_LOGIC_VECTOR ( 0 to 0 );
    clk_100MHz : in STD_LOGIC;
    done : out STD_LOGIC;
    done_1 : out STD_LOGIC;
    interrupt : out STD_LOGIC;
    reset_rtl_0 : in STD_LOGIC;
    rxbyteclkhs : out STD_LOGIC;
    status : out STD_LOGIC_VECTOR ( 31 downto 0 );
    status_1 : out STD_LOGIC_VECTOR ( 31 downto 0 );
    video_out_tdata : out STD_LOGIC_VECTOR ( 23 downto 0 );
    video_out_tdest : out STD_LOGIC_VECTOR ( 9 downto 0 );
    video_out_tlast : out STD_LOGIC;
    video_out_tvalid : out STD_LOGIC
  );
  end component design_1;
begin
design_1_i: component design_1
     port map (
      clk_100MHz => clk_100MHz,
      done => done,
      done_1 => done_1,
      interrupt => interrupt,
      mipi_phy_if_0_clk_hs_n => mipi_phy_if_0_clk_hs_n,
      mipi_phy_if_0_clk_hs_p => mipi_phy_if_0_clk_hs_p,
      mipi_phy_if_0_clk_lp_n => mipi_phy_if_0_clk_lp_n,
      mipi_phy_if_0_clk_lp_p => mipi_phy_if_0_clk_lp_p,
      mipi_phy_if_0_data_hs_n(0) => mipi_phy_if_0_data_hs_n(0),
      mipi_phy_if_0_data_hs_p(0) => mipi_phy_if_0_data_hs_p(0),
      mipi_phy_if_0_data_lp_n(0) => mipi_phy_if_0_data_lp_n(0),
      mipi_phy_if_0_data_lp_p(0) => mipi_phy_if_0_data_lp_p(0),
      mipi_phy_if_1_clk_hs_n => mipi_phy_if_1_clk_hs_n,
      mipi_phy_if_1_clk_hs_p => mipi_phy_if_1_clk_hs_p,
      mipi_phy_if_1_clk_lp_n => mipi_phy_if_1_clk_lp_n,
      mipi_phy_if_1_clk_lp_p => mipi_phy_if_1_clk_lp_p,
      mipi_phy_if_1_data_hs_n(0) => mipi_phy_if_1_data_hs_n(0),
      mipi_phy_if_1_data_hs_p(0) => mipi_phy_if_1_data_hs_p(0),
      mipi_phy_if_1_data_lp_n(0) => mipi_phy_if_1_data_lp_n(0),
      mipi_phy_if_1_data_lp_p(0) => mipi_phy_if_1_data_lp_p(0),
      reset_rtl_0 => reset_rtl_0,
      rxbyteclkhs => rxbyteclkhs,
      status(31 downto 0) => status(31 downto 0),
      status_1(31 downto 0) => status_1(31 downto 0),
      video_out_tdata(23 downto 0) => video_out_tdata(23 downto 0),
      video_out_tdest(9 downto 0) => video_out_tdest(9 downto 0),
      video_out_tlast => video_out_tlast,
      video_out_tvalid => video_out_tvalid
    );
end STRUCTURE;
```

A.2.4 Edited CSI2 Tx Subsystem Example Design VHDL Wrapper

```
--Copyright 1986-2019 Xilinx, Inc. All Rights Reserved.
----------------------------------------------------------------------------------
--Tool Version: Vivado v.2019.2 (win64) Build 2708876 Wed Nov  6 21:40:23 MST 2019
--Date         : Thu May 25 12:06:22 2023
--Host         : 5CG8505X0V running 64-bit major release  (build 9200)
--Command      : generate_target design_1_wrapper.bd
--Design       : design_1_wrapper
--Purpose      : IP block netlist
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
library UNISIM;
use UNISIM.VCOMPONENTS.ALL;
entity design_1_wrapper is
  port (
    LED : out STD_LOGIC_VECTOR ( 0 to 0 );
    mipi_phy_if_0_clk_n : out STD_LOGIC;
    mipi_phy_if_0_clk_p : out STD_LOGIC;
    mipi_phy_if_0_data_n : out STD_LOGIC_VECTOR ( 0 to 0 );
    mipi_phy_if_0_data_p : out STD_LOGIC_VECTOR ( 0 to 0 )
  );
end design_1_wrapper;

architecture STRUCTURE of design_1_wrapper is
  component design_1 is
  port (
    LED : out STD_LOGIC_VECTOR ( 0 to 0 );
    mipi_phy_if_0_clk_n : out STD_LOGIC;
    mipi_phy_if_0_clk_p : out STD_LOGIC;
    mipi_phy_if_0_data_n : out STD_LOGIC_VECTOR ( 0 to 0 );
    mipi_phy_if_0_data_p : out STD_LOGIC_VECTOR ( 0 to 0 )
  );
  end component design_1;
begin
design_1_i: component design_1
     port map (
      LED(0) => LED(0),
      mipi_phy_if_0_clk_n => mipi_phy_if_0_clk_n,
      mipi_phy_if_0_clk_p => mipi_phy_if_0_clk_p,
      mipi_phy_if_0_data_n(0) => mipi_phy_if_0_data_n(0),
      mipi_phy_if_0_data_p(0) => mipi_phy_if_0_data_p(0)
    );
end STRUCTURE;
```

References

[1] "A heritage of safety innovations." https://www.volvocars.com/intl/v/car-safety/safety-heritage. [Online]. Accessed: 2023-05-08.

[2] "What are the differences between active and passive safety features on cars?." https://www.eckellsparks.com/2022/03/02/difference-active-passive-safety-features/?utm_source=rss&utm_medium=rss&utm_campaign=difference-active-passive-safety-features, 2022. [Online]. Accessed: 2023-05-08.

[3] "Adas vs autonomous driving: What is the role of adas in autonomous driving?." https://caradas.com/adas-vs-autonomous-driving/, 2022. [Online]. Accessed: 2023-05-08.

[4] Volvo Cars, "Driver assistance systems." https://www.volvocars.com/au/v/car-safety/driver-assistance. [Online]. Accessed: 2023-05-08.

[5] Volvo Cars, "Safety: Culture and vision." https://www.volvocars.com/intl/v/safety/culture-vision, 2022. [Online]. Accessed: 2023-05-09.

[6] J. Hoffmann, "Modeling, simulation, and injection of camera images/video to automotive embedded ecu," Master's thesis, University of Stuttgart, Stuttgart, Germany, 2023. Unpublished.

[7] N. C. Basavaraju, "Modeling, simulation, and injection of camera images/video to automotive embedded ecu," Master's thesis, Uppsala University, Uppsala, Sweden, 2023. Unpublished.

[8] H. H. Bengtsson, M. Hiller, and S. Sigfridsson, "Tsn ethernet as core network in the centralized e/e architecture - challenges and possible solution." https://standards.ieee.org/wp-content/uploads/import/documents/other/eipatd-presentations/2019/D1-02_BENGTSSON-TSN_ethernet_as_core_network_in_EE_architecture.pdf, 2019. [Online]. Accessed: 2023-05-09.

[9] Kvaser, "Introduction to the lin bus." https://www.researchgate.net/publication/321935869_Radiation_Effects_on_MOSFETs, 2022. [Online]. Accessed: 2023-05-09.

[10] Millennium Circuits Limited, "Field programmable gate array (fpga) vs. microcontroller — what's the difference?." https://www.mclpcb.com/blog/fpga-vs-microcontroller/. [Online]. Accessed: 2023-05-11.

[11] Xilinx, "Cpld." https://www.xilinx.com/products/silicon-devices/cpld/cpld.html. [Online]. Accessed: 2023-05-31.

[12] javatpoint, "Difference between cpld and fpga." https://www.javatpoint.com/cpld-vs-fpga. [Online]. Accessed: 2023-05-31.

[13] Lithmee, "What is the difference between cpld and fpga." https://pediaa.com/what-is-the-difference-between-cpld-and-fpga/, 2019. [Online]. Accessed: 2023-05-31.

[14] Bae Systems, "What is a system-on-a-chip?." https://www.baesystems.com/en-us/definition/what-is-a-system-on-a-chip. [Online]. Accessed: 2023-05-11.

[15] reflexces, "Xilinx zynq ultrascale+ mpsoc." https://www.reflexces.com/modules/xilinx-zynq-ultrascale-mpsoc. [Online]. Accessed: 2023-05-11.

[16] Doulos, "Rtl coding." https://www.doulos.com/knowhow/vhdl/rtl-coding/. [Online]. Accessed: 2023-05-16.

[17] Xilinx, "Rtl-to-bitstream design flow." https://docs.xilinx.com/r/en-US/ug892-vivado-design-flows-overview/RTL-to-Bitstream-Design-Flow, 2022. [Online]. Accessed: 2023-05-16.

[18] Xilinx, "Intellectual property." https://www.xilinx.com/products/intellectual-property.html. [Online]. Accessed: 2023-05-17.

[19] Xilinx, "Amba axi4 interface protocol." https://www.xilinx.com/products/intellectual-property/axi.html. [Online]. Accessed: 2023-05-26.

[20] Xilinx, "Axi basics 1 - introduction to axi." https://support.xilinx.com/s/article/1053914?language=en_US, 2023. [Online]. Accessed: 2023-05-26.

[21] Nandland, "How serdes works in an fpga, high speed serial tx/rx for beginners." https://www.youtube.com/watch?v=3xZsTBEUkFI, 2021. [Online]. Accessed: 2023-05-25.

[22] MIPI Alliance, "Mipi camera serial interface 2." https://www.mipi.org/specifications/csi-2, 2016. [Online]. Accessed: 2023-05-26.

[23] NXP Semiconductors, "Mipi–csi2 peripheral on i.mx6 mpus." https://www.nxp.com/docs/en/application-note/AN5305.pdf, 2016. [Online]. Accessed: 2023-05-26.

[24] Flexmedia XM, "Know-how: what is gmsl2? capabilities and test challenges." https://flexmediaxm.com/know-how-what-is-gmsl2-capabilities-and-test-challenges/. [Online]. Accessed: 2023-05-25.

[25] Computerphile, "Capturing digital images (the bayer filter) - computerphile." https://www.youtube.com/watch?v=LWxu4rkZBLw, 2015. [Online]. Accessed: 2023-05-26.

[26] Wikipedia, "Bayer filter." https://en.wikipedia.org/wiki/Bayer_filter, 2023. [Online]. Accessed: 2023-05-26.

[27] Arrow, "Introduction to bayer filters." https://www.arrow.com/en/research-and-events/articles/introduction-to-bayer-filters, 2019. [Online]. Accessed: 2023-05-26.

[28] B. Fraser, "Understanding digital raw capture." https://www.adobe.com/digitalimag/pdfs/understanding_digitalrawcapture.pdf, 2004. [Online]. Accessed: 2023-05-26.

[29] Xilinx, "Zynq ultrascale+ mpsoc." https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html. [Online]. Accessed: 2023-05-15.

[30] Mouser Electronics, "Xilinx zynq® ultrascale+™ mpsoc multiprocessors." https://www.mouser.se/new/xilinx/xilinx-zynq-ultrascale-mpsocs/. [Online]. Accessed: 2023-05-15.

[31] Trenz Electronic, "Teb0911 ultrarack+ mpsoc board with amd zynq ultrascale+ zu9,6 fmc connectors." https://shop.trenz-electronic.de/en/TEB0911-04-9BEX1MA-TEB0911-UltraRack-MPSoC-Board-with-AMD-Zynq-UltraScale-ZU9-6-FMC-Connectors. [Online]. Accessed: 2023-05-11.

[32] Xilinx, "Fpga mezzanine card." https://www.xilinx.com/products/boards-and-kits/fmc-cards.html. [Online]. Accessed: 2023-05-12.

[33] Trenz Electronics, "Teb0911 trm." https://shop.trenz-electronic.de/trenzdownloads/Trenz_Electronic/Motherboards_and_Carriers/TEB0911/REV04/Documents/TRM-TEB0911-04.pdf, 2019. [Online]. Accessed: 2023-05-15.

[34] FMCHUB, “Vita 57 fpga mezzanine card (fmc) signals and pinout of high-pin count (hpc) and low-pin count (lpc) connectors.” https://fmchub.github.io/appendix/VITA57_FMC_HPC_LPC_SIGNALS_AND_PINOUT.html. [Online]. Accessed: 2023-05-15.

[35] Trenz Electronic, “Tef0007.” https://wiki.trenz-electronic.de/display/PD/TEF0007, 2018. [Online]. Accessed: 2023-05-15.

[36] Xilinx, “Kintex 7.” https://www.xilinx.com/products/silicon-devices/fpga/kintex-7.html. [Online]. Accessed: 2023-05-15.

[37] Sundance Technology, “Tef0007-02a.” https://www.sundance.technology/system-on-modules-som/fmc-modules/io-modules/tef0007-02a/. [Online]. Accessed: 2023-05-15.

[38] Trenz Electronic, “Tef0010.” https://wiki.trenz-electronic.de/display/PD/TEF0010, 2020. [Online]. Accessed: 2023-05-15.

[39] Xilinx, “Artix-7.” https://www.xilinx.com/products/silicon-devices/fpga/artix-7.html. [Online]. Accessed: 2023-05-16.

[40] Sundance Technology, “Tef0010-01.” https://www.sundance.technology/system-on-modules-som/fmc-modules/io-modules/tef0010-01/. [Online]. Accessed: 2023-05-15.

[41] Peter Krogh, “Camera raw images.” https://www.mediagraph.io/blog/camera-raw-images, 2020. [Online]. Accessed: 2023-05-15.

[42] Trenz Electronics, “Teb0911.” https://shop.trenz-electronic.de/trenzdownloads/Trenz_Electronic/Motherboards_and_Carriers/TEB0911/REV04/Documents/AD-TEB0911-04-9BEX1FA.PDF. [Online]. Accessed: 2023-05-30.

[43] Xilinx, “Downloads.” https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vivado-design-tools/archive.html. [Online]. Accessed: 2023-05-17.

[44] Trenz Electronics, “Teb0911 ultrarack+ mpsoc board with amd zynq™ ultrascale+™ zu9,6 fmc connectors.” https://shop.trenz-electronic.de/en/TEB0911-04-9BEX1MA-TEB0911-UltraRack-MPSoC-Board-with-AMD-Zynq-UltraScale-ZU9-6-FMC-Connectors?path=Trenz_Electronic/Motherboards_and_Carriers/TEB0911/Reference_Design. [Online]. Accessed: 2023-05-15.

[45] Xilinx, “Artix-7 fpgas data sheet: Dc and ac switching characteristics.” https://docs.xilinx.com/v/u/en-US/ds181_Artix_7_Data_Sheet, 2022. [Online]. Accessed: 2023-05-16.

[46] Xilinx, “Logic synthesis.” https://www.xilinx.com/products/design-tools/vivado/implementation.html#synthesis. [Online]. Accessed: 2023-05-17.

[47] Xilinx, “Implementation.” https://www.xilinx.com/products/design-tools/vivado/implementation.html#implementation. [Online]. Accessed: 2023-05-17.

[48] Xilinx, “Vivado design suite user guide using constraints.” https://www.xilinx.com/support/documents/sw_manuals/xilinx2022_1/ug903-vivado-using-constraints.pdf, 2022. [Online]. Accessed: 2023-05-31.

[49] Xilinx, “Vitis unified software platform documentation: Embedded software development (ug1400).” https://docs.xilinx.com/viewer/book-attachment/PHwa3sDuk6G_sjX_v9CnHw/4Yl0KRfcZ5H6NcC7s7Ye9g, 2023. [Online]. Accessed: 2023-05-17.

[50] Xilinx, “Mipi csi controller subsystems.” https://www.xilinx.com/products/intellectual-property/ef-di-mipi-csi-rx.html. [Online]. Accessed: 2023-05-18.

[51] Xilinx, “Mipi csi-2 transmit subsystem v2.0 product guide.” https://docs.xilinx.com/v/u/2.0-English/pg260-mipi-csi2-tx, 2019. [Online]. Accessed: 2023-05-18.

[52] Xilinx, “Clocking wizard logicore ip product guide (pg065).” https://docs.xilinx.com/viewer/book-attachment/eBBMExOBcxQAxRvH9B~~uA/FpyRRAcCZJJEmbXGsBhjXQ, 2022. [Online]. Accessed: 2023-05-18.

[53] Xilinx, “Processor system reset module v5.0 product guide (pg164).” https://docs.xilinx.com/v/u/en-US/pg164-proc-sys-reset, 2015. [Online]. Accessed: 2023-05-18.

[54] Xilinx, “Axi traffic generator v3.0 product guide (pg125).” https://docs.xilinx.com/v/u/en-US/pg125-axi-traffic-gen, 2019. [Online]. Accessed: 2023-05-18.

[55] Xilinx, “Vivado design suite user: Guide designing with ip.” https://www.xilinx.com/support/documents/sw_manuals/xilinx2021_2/ug896-vivado-ip.pdf, 2021. [Online]. Accessed: 2023-05-19.

[56] Xilinx, “Axi interconnect v2.1 logicore ip product guide (pg059).” http://users.ece.utexas.edu/~mcdermot/arch/articles/Zynq/pg059-axi-interconnect.pdf, 2016. [Online]. Accessed: 2023-05-18.

[57] Xilinx, “Video test pattern generator v8.0 logicore ip product guide (pg103).” https://docs.xilinx.com/v/u/8.0-English/pg103-v-tpg, 2019. [Online]. Accessed: 2023-05-18.

[58] Lattice Semiconductor, “Crosslink.” https://www.latticesemi.com/Products/FPGAandCPLD/CrossLink. [Online]. Accessed: 2023-05-20.

[59] Lattice Semiconductor, “Lattice diamond programmer and deployment tool.” https://www.latticesemi.com/programmer. [Online]. Accessed: 2023-05-20.

[60] Xilinx, “Zynq ultrascale+ mpsoc processing system v2.0 product guide.” https://docs.xilinx.com/v/u/2.0-English/pg201-zynq-ultrascale-plus-processing-system, 2016. [Online]. Accessed: 2023-05-22.

[61] Xilinx Forum, “What ’s the define of the free run clock.” https://support.xilinx.com/s/question/0D52E00006hpcV8SAI/what-s-the-define-of-the-free-run-clock?language=en_US, 2017. [Online]. Accessed: 2023-05-23.