System integration of image processing and further developing of CCSDS and integration of RS422

Armin Bahadoran

NTNU
Fakultet for informasjons-
teknologi og elektroteknikk
Institutt for elektroniske systemer

Abstract

When sending data onboard a satellite, the data transmission rate between modules is an issue. This thesis describes how to incorporate a UART data transmission function into a Zynq-based System on Chip. The Zynq is a System on Chip that connects a processor with programmable logic. The data is sent via the processing system, although the programmable logic uses a UART device. The functionality will be to use interrupts to transport data from the OPU to the PC. To ensure proper operation, tests were carried out using stimuli from the processing system and monitored using a protocol analyzer.

Summary

When hyperspectral pictures are collected and compressed, they must be stored in the satellite's storage device, i.e. the payload controller, which also serves as a router between the payload and the satellite bus. In order to communicate between its Onboard Processing Unit and the Payload Controller, the HYPSO-1 satellite, which is presently in orbit, uses the CAN standard. This method works fine, however data transmission takes some time. The NTNU Small Satellite Team has concluded that in parallel with the CAN interface, a UART coupled to an RS-422 with a baud rate close to 3 Mbps is preferable. Implementing an AXI UART 16550 peripheral to transport data from the memory to the Payload Controller is the method which is decided. This is accomplished by transferring serially with the peripheral the compressed HSI data from the Cube DMA, which is kept in a data buffer. The latency of the UART functionality is reduced by utilizing interrupts. The data is delivered from the device's transmit line to the RS-422, which is further linked to the Payload Controller. After integration and testing, the UART operated as intended, with data being successfully transferred and received between two devices. To complete the project, the UART driver must be made compatible with embedded Linux so that it can function with the other Onboard Processing Unit programs.

Preface

This report is written by Armin Bahadoran as the end of the 2 year master programme named Electronic Systems Design with the specialization in Design of Digital Circuits. This master's project was published by the NTNU Small Satellite Team, a group working on satellite related activities.
The NSSL is now working on HYPSO-2, a nanosatellite with a payload that includes a hyperspectral imager (HSI) camera and also a software defined radio (SDR). On the satellite, there was no UART device at the start of this project.
The goal of this research is to show how compressed hyperspectral imaging (HSI) data may be transmitted and received using the UART, which has been implemented into the onboard processing unit.

Acknowledgements

First and foremost, I'd like to thank my thesis adviser, Dr. Milica Orlandic of NTNU's Department of Electronic Systems. I want to thank her for all of her help and advice during the endeavor. She continuously let me do my own work on this project, but she pointed me in the appropriate path when she believed I needed it.

I'd also want to thank all of the NSSL members. During the pandemic, the ability to borrow lab equipment was incredibly useful. Also, for the sake of motivation, the members' advice and sympathy when times were rough were highly helpful. The team also allowed me to experience numerous work methodologies, which will be highly beneficial when I begin to work.

Finally, I want to express my sincerest appreciation to my parents and friends (in particular, an exchange student who was always there for me when things became difficult) for their unwavering support and encouragement during my years of study and the process of researching and writing my thesis. When circumstances were bad, you always found a way to keep me motivated. This achievement would not have been achievable without their assistance.

Contents

List of Figures

List of Tables

Abbreviations

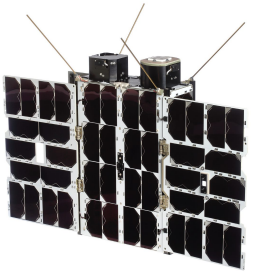
LEO Low Earth Orbit
HYPSO Hyperspectral Smallsat for Ocean Observation
FPGA Field-Programmable Gate Array
OPU OnBoard Processing Unit
PLL Phase Locked Loop
DDR Double Data Rate
PS Processing System
PL Programmable Logic
UAV Unmanned Aerial Vehicle
RISC Reduced Instruction Set Computer
CISC Complex Instruction Set Computer
RTOS Real-Time Operating System
CSP Cubesat Space Protocol
POSIX Portable Operating System Interface
MIO Multiplexed I/O
MPSoC Multiprocessor System on a Chip
UHF Ultra High Frequency
KDD Kernel Device Driver
BSP Board Support Package
USB Universal Serial Bus
ESL Electronic System Level
DMA Direct Memory Access
UART Universal Asynchronous Receiver-Transmitter
CCSDS Consultative Committee for Space Data Systems
NASA National Aeronautics and Space Administration
BIP Band Interleaved by Pixel
BSQ Band Sequential
MM2S Memory Map to Stream
S2MM Stream to Memory Map
NTNU Norwegian University of Science and Technology
SDR Software Defined Radio
NSSL NTNU Small Satellite Lab
ROM Payload Controller
FSBL First Stage Bootloader
U-Boot Second Stage Boot Loader
FIFO First-In, First-Out
OCM On-Chip Memory
RTL Receive Trigger Level
CPU Central Processing Unit

RS-422 Recommended Standard 422
HSI Hyperspectral Imaging
RGB Red Green Blue
FC Flight Computer
PC Payload Controller
BoB Breakout Board
EPS Electronic Power System
CAN Controller Area Network
SoM System on Memory
ILA Integrated Logic Analyzer
VIO Virtual Input/Output
OS Operating System
ISR Interrupt Service Routine
AXI Advanced eXtensible Interface
CLK Clock
IP Intellectual Property
PMOD Peripheral Module
Tcl Tool Command Language
Tx Transmit
Rx Receive
SoC System on Chip
IRQ Interrupt Request
GIC General Interrupt Controller

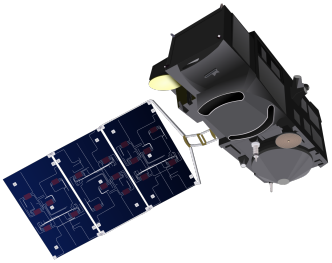
Chapter 1

Introduction

Thousands of artificial satellites orbit Earth. These satellites have various different purposes, some do collect pictures of the planet to help meteorologists predict the weather and track hurricanes. Some take pictures of other planets, the sun, black holes, dark matter, or faraway galaxies. These pictures assist scientists in better understanding the solar system and universe [26]. The bird's eye view that satellites have allowed for a large collection of data, more quickly than what instruments on the ground would achieve. One of the most important factors with satellites projects is the cost of the launch. NASA's space shuttle had a cost of about $1.5 billion to launch 27,500 kg to Low Earth Orbit (LEO). Today SpaceX's Falcon 9 announce a cost of $62 million to launch 22,800 kg to LEO [27]. Small satellites have revolutionized access to space by dramatically reducing the cost of launching and operating a satellite while in space. This has allowed for new opportunities for universities, the commercial sector, and national space agencies [28]. Still, there do exist issues being limited hardware access, shorter lifespan due to rapid orbital decay, and lower transmitter output signal than traditional satellites.



(a) 6U nanosatellite bus M6P (1 - 10 kg) [5]



(b) ESA Sentinel-3A (> 500 kg) [29]

Figure 1.1: Comparison between M6P and Sentinel-3A

1.1 HYPSO Mission

At the Norwegian University of Science and Technology (NTNU) an organization named NTNU Small Satellite Lab (NSSL) consisting of bachelor-, graduate students, PhDs, and Ph.D. students have created a goal to create a small satellite to be sent to space for observing oceanographic phenomena. The mission is named Hyperspectral Smallsat for Ocean Observation (HYPSO) with the main objective to observe the ocean through a hyperspectral camera. The camera allows for the capture of a vast number of contiguous spectral bands across the electromagnetic spectrum. This is used to be able to discover harmful algae bloom in the Norwegian Sea in order to avoid fish die-off. The launch of the satellite will be in the year 2024. The satellite will be equipped with a hyperspectral- and an RGB camera. By the usage of hyperspectral imaging (HSI), it's possible to investigate bands of light frequencies and focus on designated colors [30].

The HYPSO project's satellite will be a CubeSat. The CubeSat's size is measured in cubes, i.e. **Us**, where 1U is 10 cm x 10 cm x 10 cm [31]. The satellite for this project will be 6U provided by NanoAvionics LLC (Limited Liability Company), with dimensions of 340.5 mm x 226.3 mm x 100 mm (LxWxH) [32].

Onboard the satellite, there are effectively two missions going on: the SDR-mission (Software Defined Radio) and the HSI-mission (Hyper Spectral Imager). The SDR mission's long-term purpose is to develop Arctic communication infrastructure to enable data retrieval from various sensor nodes and robotic agents [32]. This project is about the HSI- mission, also known as the HYPSO (HYPer-spectral Smallsat for Ocean Observation). The mission will conduct ocean observations, especially gathering hyperspectral data from the ocean's surface with a hyperspectral image and sending the data to Earth [30].

The Figure 1.2 on the next page shows the HYPSO mission's operational concept as well as the stages of a satellite fly-by over the Norwegian Sea. When the satellite is not in use, it goes into sleep mode. When it wakes up, it contacts the ground station located in Trondheim to receive orders, afterwards it gathers hyperspectral images of the ocean and connects to a ground station located in Svalbard to relay the data. It may take many fly-bys before all of the data is relayed, and several ground stations might be employed as data receiving stations. An unmanned aerial vehicle (UAV) travels over the same region as the satellite to gather reference photos that may be utilized to improve algal identification and atmospheric correction algorithms.

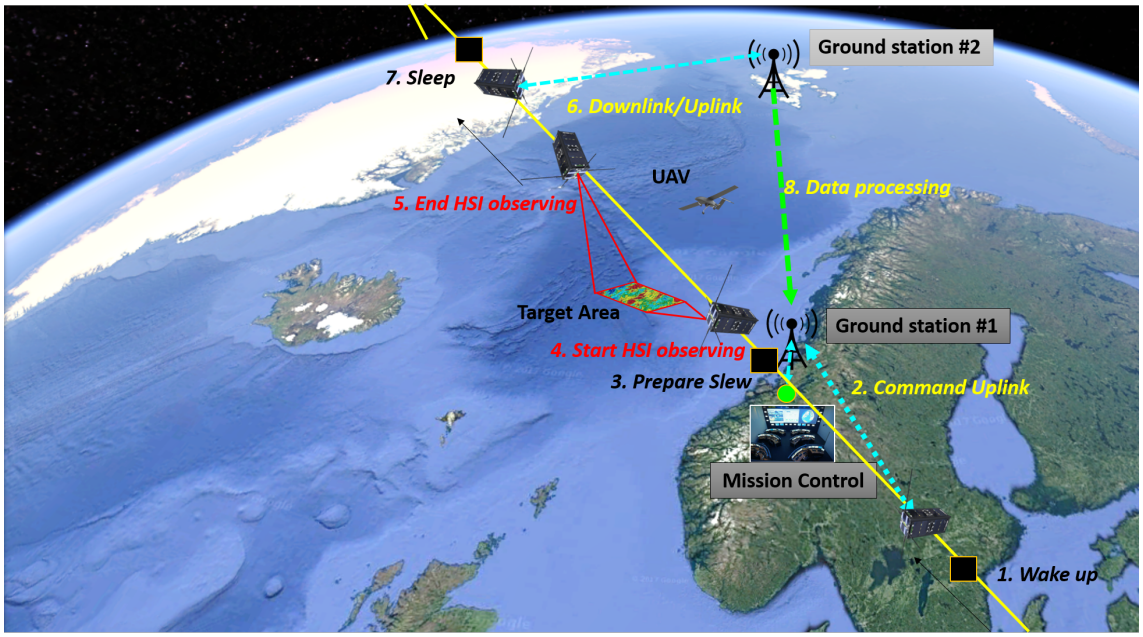


Figure 1.2: Different stages of the mission process HYPSO-1 [1]

The satellite shown in Figure 1.3 will include a star tracker and a Red, Green, Blue (RGB) camera which does point at the direction of the Earth, which are used to assist in determining the spacecraft's orientation, while the orientation of the satellite is controlled by reaction wheels and magnetorquers. The mission data from the HSI is to be processed by the OPU (Onboard Processing Unit).

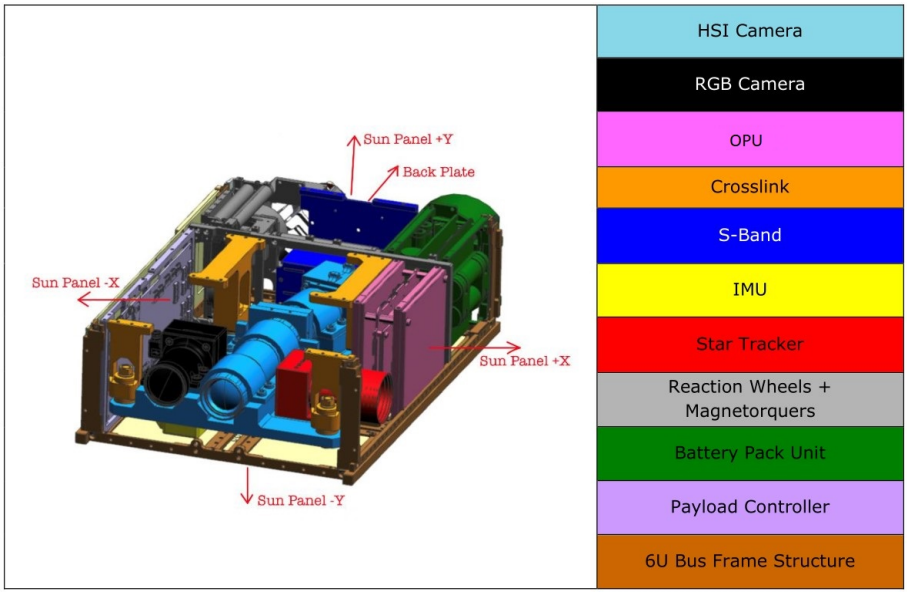


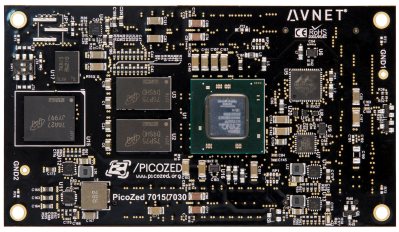
Figure 1.3: HYPSO-1 CubeSat architecture, inspired from [2]

Be aware that Figure 1.3 does show the CubeSat architecture of the HYPSO-1 system, and not the HYPSO-2. This does not include the SDR module, which is to be integrated into HYPSO-2 beside the OPU module.

1.2 Issue

One main problem with hyperspectral images is that they can be up to 140 megabytes [33]. As a result, the transfer time between the different modules onboard the satellite is extensive. Thus the need to reduce the file size is essential. This is done through the usage of a compression algorithm for high-speed computation. In order to achieve this, the team has decided to use a Multiprocessor System on a Chip (MPSoC) with a Field-Programmable Gate Array (FPGA) built inside. The HYPSO-2 mission will be using a Xilinx Zynq UltraScale+ MPSoC for the onboard processing unit (OPU) instead of the previous board which was the PicoZed 7030 using Zynq-7000 All Programmable System on Chip.

(a) UltraZed-EV SOM Top View [11]



(b) PicoZed 7030 SOM Front View [34]

Figure 1.4: Different Processing Boards

Implementing the compression algorithm on the satellite it will allow for faster download time. From this issue, a group named the Consultative Committee for Space Data Systems has been able to develop the standard CCSDS-123 for compression of hyperspectral images. A previous member of the small satellite team implemented the CCSDS123 onto the FPGA for his master thesis to be used on the OPU in the HYPSO satellite [21]. This thesis will take into use what has already been developed, by integrating the technical standard Recommended Standard (RS-422) into the system.

1.3 Goal

In HYPSO-1's system architecture, the spacecraft does its data transfer through the Controller Area Network (CAN) serial communication protocol. The specialization project done in the previous semester was an introduction to the work to be done in this master thesis. The work consisted of creating a UART protocol from scratch [35]. This project was not integrated into the OPU-system being that it does not interface with the processing system, still, it is a good document to read to become familiar with the protocol itself, making this paper more understandable. The goal of this project is to implement the AXI UART 16550 v2.0 Intellectual Property [22] into the OPU to allow for faster data transfer between the OPU and Payload Controller (PC).

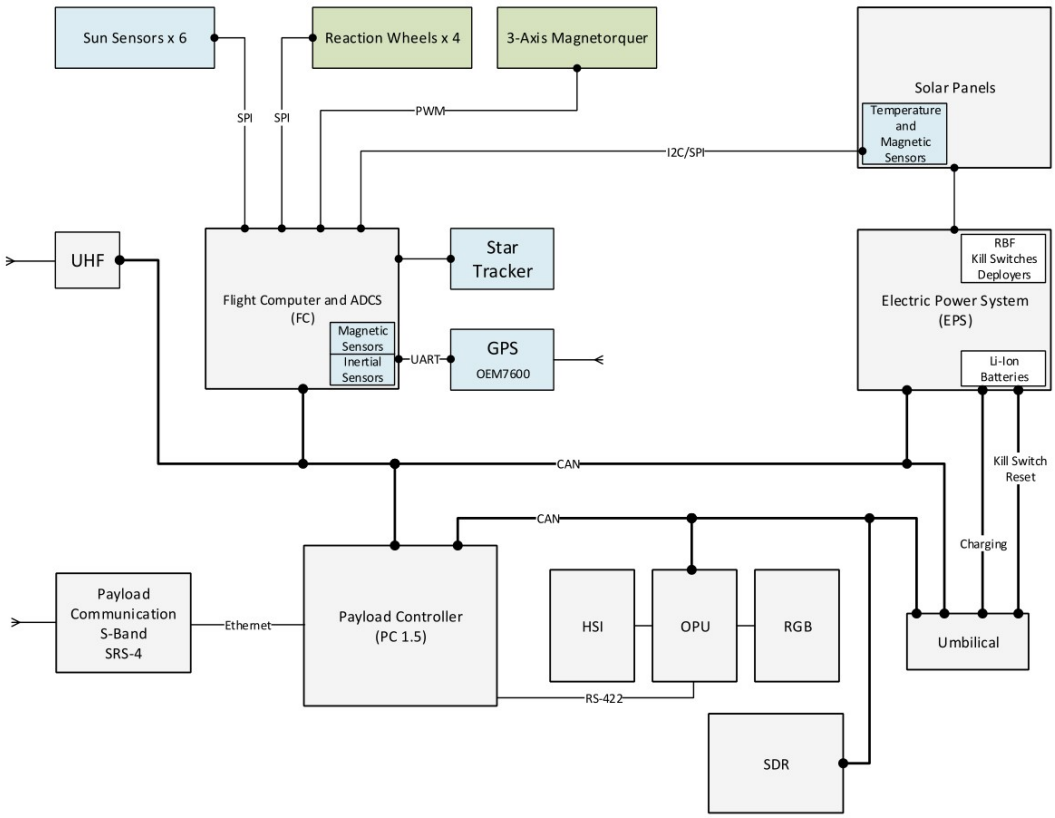


Figure 1.5: HYPSO-2 Satellite Architecture [3]

1.3.1 Thesis structure

The prototyping, implementation, integration, and testing of UART capabilities within the Onboard Processing Unit will be covered in this paper.

- **Chapter 1** Introduction
- **Chapter 2** Specialization Project
- **Chapter 3** Background
- **Chapter 4** Implementation
- **Chapter 5** Testing
- **Chapter 5** Discussion
- **Chapter 6** Conclusion

The first part consisting of the first 3 chapters will introduce the problem, the current satellite system, and relevant theory. The outcomes of the implementation, integration, and testing solutions will be presented in the second section. The last section will go through discussion and conclude whether or not the problem has been solved.

Chapter 2

Specialization Project

For the specialization project, the task was to implement and integrate a UART communication protocol to the OPU system on the Zynq UltraScale+ MPSoC EV. The RTL code was fully written in VHDL, the goal of the module was to be able to transmit and also receive data from the external source Payload Controller through an RS-422 standard. The baud rate was expected to be between 3 and 4 Mbps. Both the transmitter and receiver hardware design can be found in Appendix C.1 and Appendix C.3 respectively as well as their test benches C.2 and C.4.

Both the transmitter and the receiver part of the design were tested in the Vivado simulation window, and also on the Xilinx Zynq-7000 All Programmable (AP) SoC mounted on the ZedBoard development kit on hardware i.e. sending data from the personal computer to the Zynq-7000 in real-time through the usage of the Xilinx VIO (Virtual Input/Output) IP [36] also using the Xilinx ILA (Integrated Logic Analyzer) IP [37] to examine the wave diagram.

2.1 Verification of Design

As previously stated, the hardware verification of the design was performed in Vivado's IP Integrator window, using both ILA and VIO IPs to test both the transmitter and receiver functionality of the UART design.

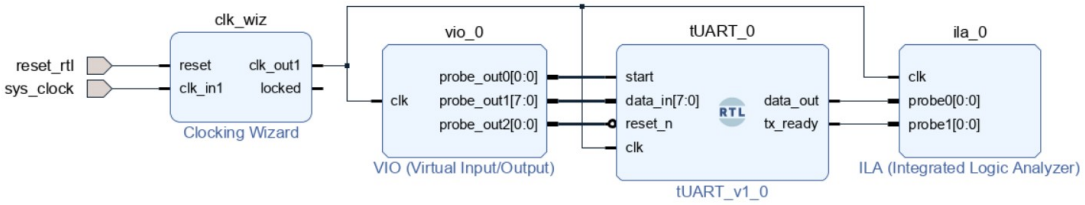


Figure 2.1: Test connection between transmitter design and IPs from Xilinx

2.1.1 Integrated Logic Analyzer (ILA)

The Integrated Logic Analyzer (ILA) IP core is a logic analyzer core that may be used to monitor a design's internal signals. Many sophisticated capabilities of current logic analyzers, such as boolean trigger equations and edge transition triggers, are included in the ILA core [37].

2.1.2 Virtual Input/Output (VIO)

The LogiCORE IP Virtual Input/Output (VIO) core is a programmable core that can monitor and control internal FPGA signals in real-time. To interact with the FPGA design, the number and width of the input and output ports may be customized [36].

2.1.3 Results from the design

For the verification of the design, there was made a test bench which gave the transmitter module information about sending the ASCII information "A L G" in the form of a hex value which is 0x41, 0x4C, 0x47 respectively. This converted to binary is the following:

- A → 0100 0001
- L → 0100 1100
- G → 0100 0111

Sending the signal 'A' onto the chip i.e. 0x41 the output data should become 0100 0001. This is confirmed by observing the ILA waveform as can be seen in Figure 2.2

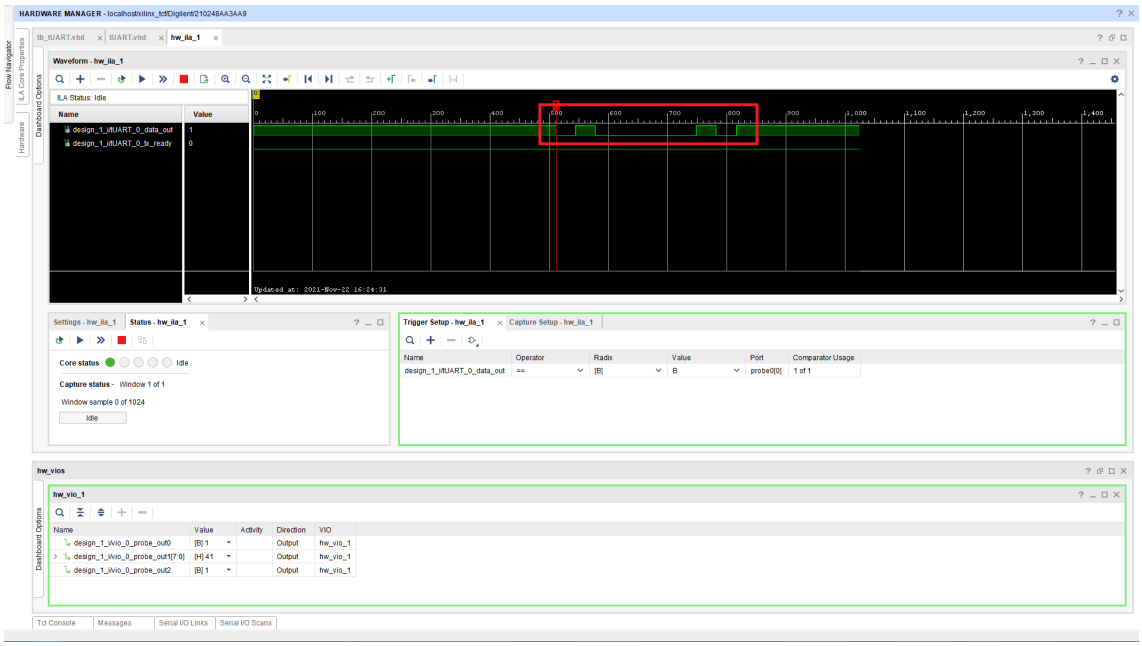


Figure 2.2: ILA waveform *data_out* result from sending character 'A' to the transmitter

The same test procedure was done to the receiver part of the project with a satisfactory result. Being that the design is purely hardware and not interconnecting with software it was decided to not integrate this into the OPU design. Still, the design of the UART has allowed for an easier understanding of the protocol, please do look into "Exploration and implementation of communication protocols for satellite payload systems" [38] and "UART Design Document HYPSO-RP-003" [35] for more detail.

Chapter 3

Background

The goal of this chapter is to present the current satellite system while also providing essential background material such as hyperspectral imaging to assist the reader comprehend the methods used. How the AXI UART 16550 v2.0 LogiCORE IP works, the satellite platform itself and also the recent replacement of the PicoZed, UltraZed-EV SOM. Also the Linux kernel, drivers and memory in Linux is described. Also included is the FPGA accelerator that was previously constructed by students as part of the HYPSO project.

3.1 Hyperspectral Imaging

Hyperspectral imaging makes use of data from hundreds of wavelengths across the electromagnetic spectrum, whereas a standard RGB camera, collects single frames with information about red-green-blue. It is designed to determine the spectrum of each pixel in an image of a scene in order to detect objects, identify materials, or identify processes.

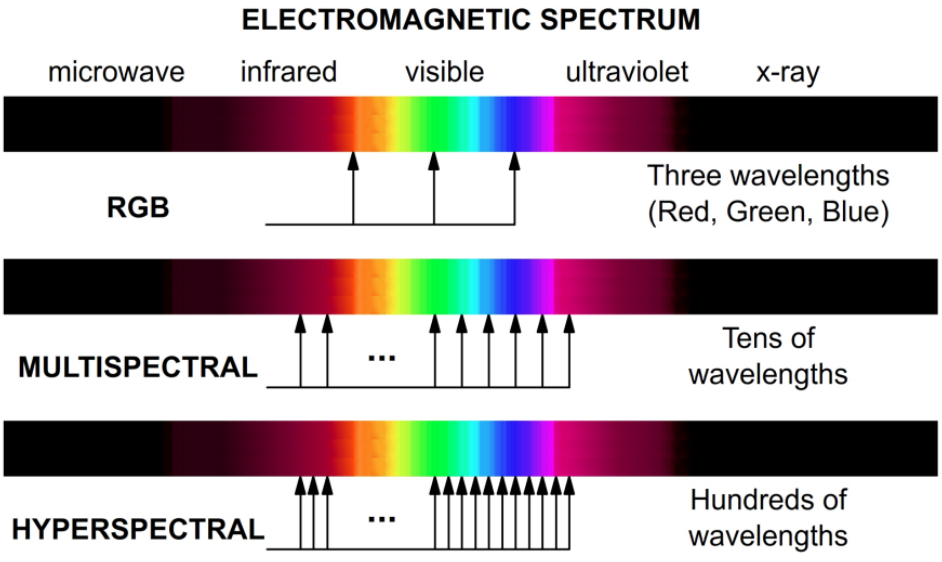


Figure 3.1: From RGB to hyperspectral [4]

3.2 HYPSO 6U Nanosatellite bus M6P

The HYPSO nanosatellite is a sophisticated assembly of parts and software from multiple student projects in collaboration with NanoAvionics. NanoAvionics develops nanosatellite casings that integrate features such as a Flight Computer (FC), a Payload Controller (PC), and intercommunication.

NSSL is developing a payload that will be integrated into the nanosatellite. The Figure 3.2 depicts the NanoAvionics satellite shell. The CSP (Cubesat Space Protocol) protocol is used by all systems and subsystems.

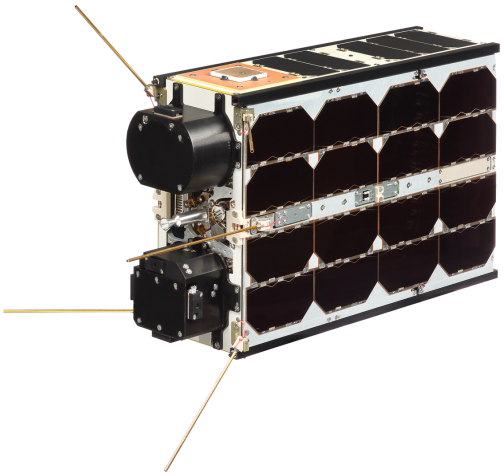


Figure 3.2: The 6U nanosatellite bus M6P [5]

The implementation of CSP is written in C and has been adapted to Free Real-Time Operating System (RTOS), Portable Operating System Interface (POSIX), and pthreads-based operating systems like Linux [39]. CSP has a service-oriented network structure divided into two segments: ground and space. The space segment is represented in Figure 1.5, while the ground segment is represented by the ground station interacting with the satellite. A CSP node is operated on each module in the network, and each node can execute several services, as well as receive different commands and data. CSP packets can be used to communicate freely across the modules.

CSP packets are packed and transferred between segments through the CAN bus or networks equipped with the Ultra High Frequency (UHF) module. The platform is equipped with two CAN busses, one for payloads and the other for satellite operation systems. The PC (Payload Controller) distributes CSP packets across CAN buses and buffers mission data from payloads before sending it to the S-band, which then transfers data to the ground station. The FC (Flight Computer) is in charge of logging telemetry data as well as attitude determination and control [40]. The satellite is powered by the EPS (Electrical Power System). The power source is Lithium-Ion batteries, which are charged by solar panels.

3.3 HSI as a Remote Sensing Tool

A hyperspectral sensor collects information as a set of images. A spectral band, also known as a wavelength range, corresponds to each image in the electromagnetic spectrum. These images are then combined into a three-dimensional (x, y, λ) hyperspectral data cube for processing and analysis, where x and y represent two spatial dimensions of the scene, and λ represents the spectral dimension [41].

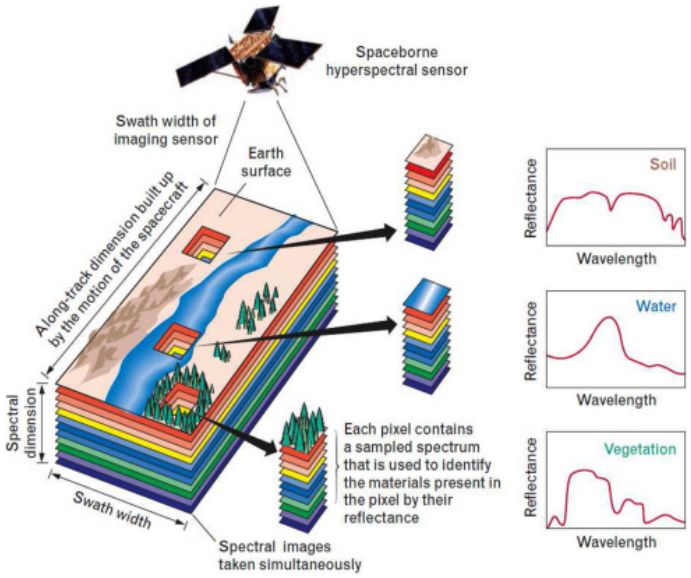


Figure 3.3: HSI concept in the remote sensing applicative context [6]

Through the use of spectral information, objects and materials can be detected with a much higher degree of precision than with a conventional red-green-blue (RGB) camera.

3.4 Remote Sensing Techniques

The HSI-cube can be obtained using different techniques. The push-broom and whiskbroom scanning methods are the two most common methods for pointing, as shown in Figure 3.4.

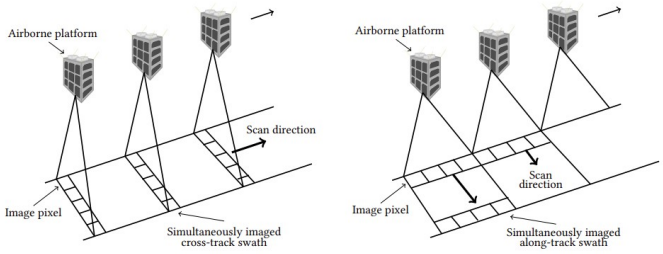


Figure 3.4: The different scanning techniques. Pushbroom on the left and whiskbroom to the right, based on the illustration from [7]

In order to accomplish its mission, HYPSO uses push-broom scanning. The image to the left in Figure 3.4 shows the camera scanning a line of light as it points

downwards. Using this line of light, the light is dispersed into several bands, thus creating a frame as shown in Figure 3.5. These frames end up being placed on top of each other, resulting in the HSI cube shown in Figure 3.6.

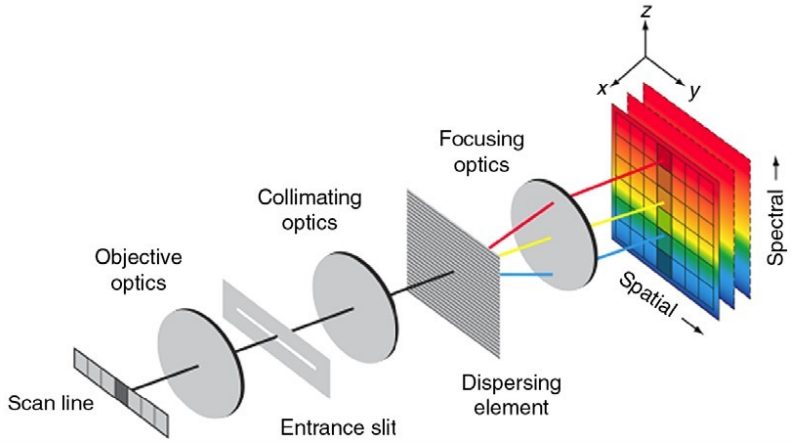


Figure 3.5: Optical chain of a common push-broom imaging spectrometer [8].

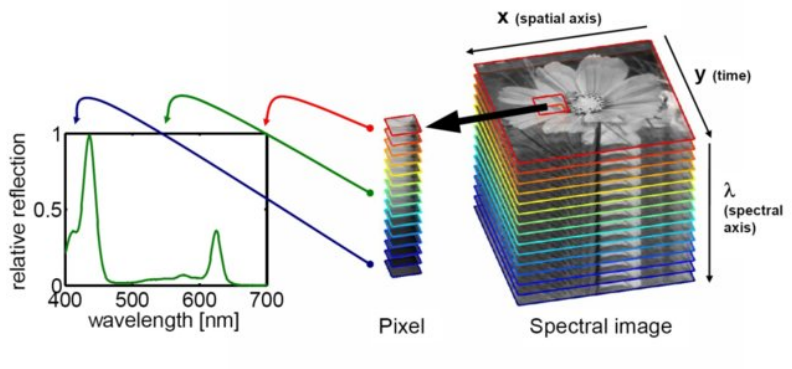


Figure 3.6: Hyperspectral datacube [9]

There are three types of ways to store hyperspectral images. Band Interleaved by Line (BIL), Band Interleaved by Pixel (BIP), and Band Sequential (BSQ) as presented in Figure 3.7

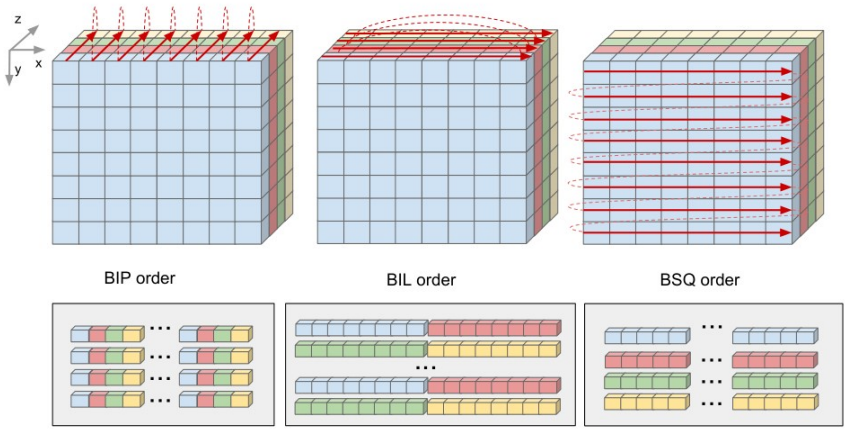


Figure 3.7: Sample ordering of HSI cube [10]

BIL sequentially stores pixels from the 'x' spatial direction for each band in each row, while the 'y' spatial direction represents the rows. The 'y' spatial direction likewise represents the rows in BIP, but inside the rows, all the bands for one pixel in the 'x' spatial direction are positioned sequentially, followed by the same for the following pixel. BSQ differs from the other two in that it represents all of the pixels in the 'x' spatial direction from one band in the columns. The 'y' spatial direction is displayed in successive rows for one band, followed by the next band [10].

3.5 Payload

The payload is comprised of the HSI, RGB cameras, and OPU.

1. **Hyperspectral camera** The HSI camera can be located in the center of the nanosatellite. It is easily identified by its size and slanted inclination as shown in Figure 1.3.

2. **RGB camera** The RGB camera is positioned on the left side of the HSI camera see Figure 1.3. Its function will be to capture the same region as the HSI camera in order to augment the hyperspectral image.

3.5.1 Onboard processing unit

Overview

The OPU system will ideally consist of a Zynq UltraScale+ MPSoC EV and a Breakout Board (BoB) connected together thus referred to as UltraBoB.

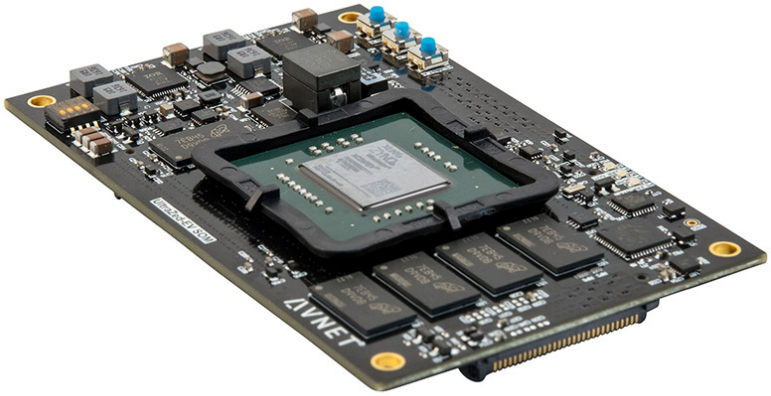


Figure 3.8: UltraZed-EV SOM Angle View [11]

This module collects and processes data from sensors mounted onboard the payload, primarily the hyperspectral imaging sensor and RGB camera. Most of the modules on the spacecraft are off-the-shelf products prepared by the Aviation & Aerospace industry NanoAvionics, except for the payload consisting of the OPU, HSI, and RGB cameras. These modules are the interest area for the other work fields in the HYPSO research project.

The HYPSO-1 architecture mostly used Controller Area Network (CAN) communication between its modules using the CubeSatProtocol (CSP) both for transmitting instruction data between the modules, and also to transmit payload data.

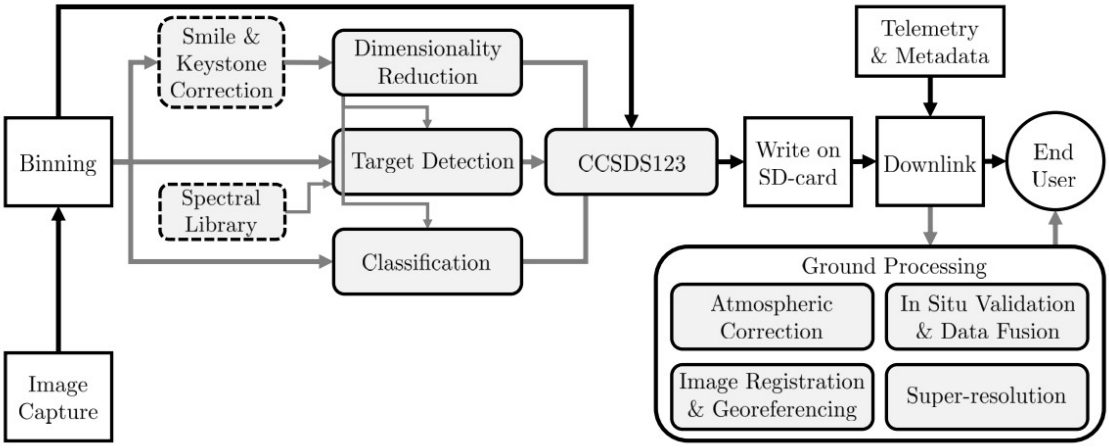


Figure 3.9: HYPSO-1 Onboard image processing pipelines [12]

The suggested onboard image processing pipelines are depicted as a block diagram, shown in Figure 3.9. The hyperspectral pictures are taken, binned, processed at a predetermined level, saved on an SD card, and downlinked together with telemetry and metadata. Depending on the data product selected, additional ground-based processing and fine-tuning might be performed before dissemination to end users. The black lines represent the simplest onboard processing pipeline, while the gray arrows provide alternate paths for customised data products [12].

The HYPSO-2 system is designed to optimize system performance by integrating Ethernet communication and adding RS-422 serial communication between OPU and PC see Figure 1.5, which will be the major focus of this thesis.

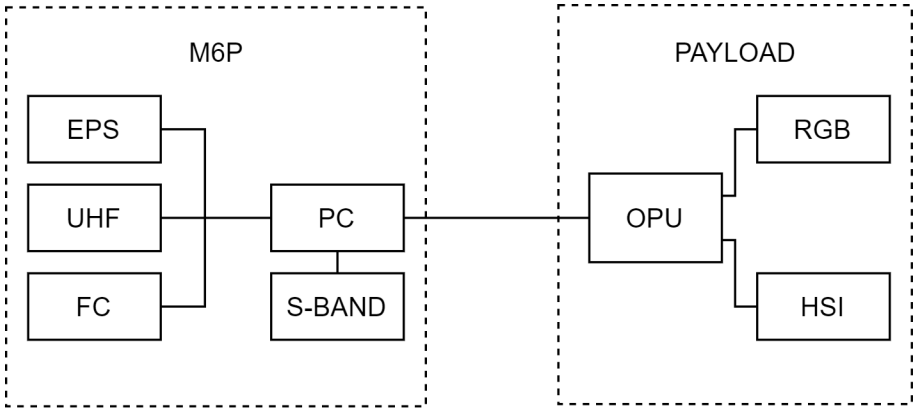


Figure 3.10: Relations between the subsystems of the HYPSO-2 satellite. Adapted diagram from [13]

The OPU is powered by the Electronic Power System (EPS), which may be managed via the Payload Controller (PC), allowing for power reset if the OPU ceases to react.

Printed Circuit Board - UltraBoB

The UltraBoB will consist of an UltraZed-EV SoM and the BoB to interface mechanically, electrically, and thermally between the M6P satellite platform and payload instruments and UltraZed-EV SoM. The UltraZed-EV is manufactured by Avnet and consists of a Xilinx XCZU7EV-1FBVB900 commonly referred to as Zynq UltraScale+ MPSoC a Dual QSPI Flash 64MB, 8GB, x8 eMMC Flash, 4GB DDR4 SDRAM PS (Processing System) and 1GB DDR4 SDRAM PL (Programmable Logic), USB (Universal Serial Bus) 2.0 and Gigabit Ethernet PHY (Physical Layer) interface controller [24].

Model	IOBs	LUTs	FF	BRAMs	URAM	DSPs
XCZU7EV	28800	230400	12720	312	96	1728

Table 3.1: Zynq UltraScale+ MPSoC: EV Device Feature Summary [24]

Interface connections, SD-Card readers, voltage regulators, and logic level shifters are all part of the BoB (Break out Board). The team's PCB engineer's 3D depiction of the BoB V3 may be seen in Figure 3.11.

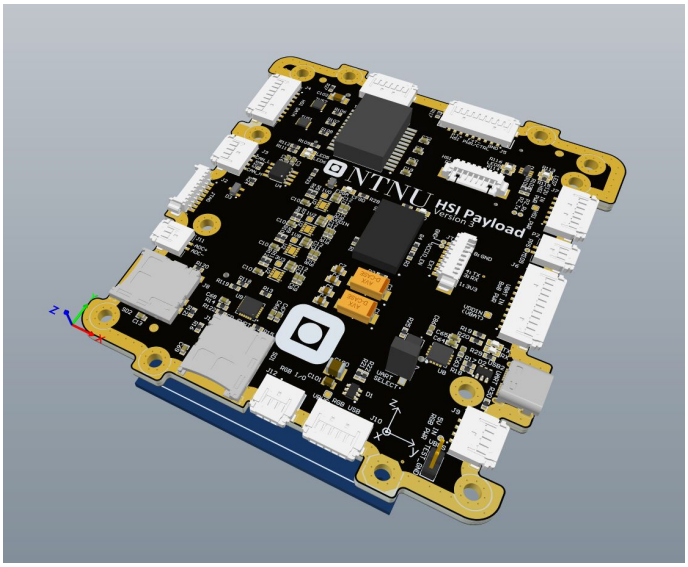


Figure 3.11: 3D rendering of HSI BoB V3 [14]

3.6 RS-422

The Electronic Industries Alliance developed RS-422, a technical standard that describes the electrical properties of a digital signaling circuit. Data transmission speeds of up to 10 Mbit/s are possible with RS-422 systems [42]. The RS-422 standard defines differential signaling, with each data line coupled with its own return line. The mark and spacing are determined by the voltage differential between these two lines. The advantage of employing a differential signal over a single ended signal is that it is more noise resistant, allowing for longer cable lengths. The signals have a voltage range of +/- 2.0V [15].

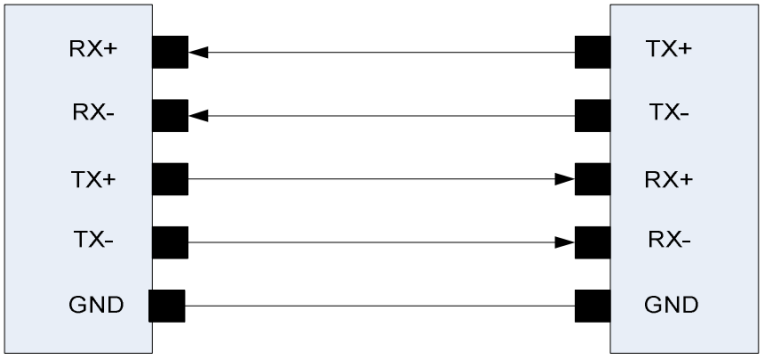


Figure 3.12: Typical RS-422 Interconnect [15]

3.7 Field Programmable Gate Array

A Field Programmable Gate Array (FPGA) is an integrated circuit with programmable logic blocks. This enables low-level programming of a chip after it has been manufactured. Unlike a CPU, the FPGA's power rests in its capacity to do many distinct particular jobs simultaneously. A bitstream file may be used to simply update the logic of the FPGA. As a result of its flexibility to be reconfigured while in use, FPGAs are a popular choice for sectors such as aerospace and nanosatellites. FPGA logic may be designed using Hardware Description Languages (HDLs) such as Verilog and VHDL [43].

3.7.1 UltraZed EV SoM

The UltraZed is a System on Module (SoM) based on Xilinx's Zynq UltraScale+ Multiprocessor System on a Chip (MPSoC). As seen in Figure 3.8.

The MPSoC utilizes four ARM Cortex-A53 CPUs in conjunction with a Field Programmable Gate Array (FPGA). The ARM processors employ the Reduced Instruction Set Computer (RISC) architecture, which, as the name indicates, is less complex than the Complex Instruction Set Computer (CISC). It reduces the number of instructions required per job while increasing the number of cycles required for each instruction. It is designed to be more efficient by employing fewer instruction sets and simpler instructions, resulting in reduced energy consumption [44].

The Zynq MPSoC's advantage stems from its utilization of both CPUs and an FPGA. The FPGA enables specialized low-level programming and parallel data processing. The Zynq UltraScale+ MPSoC is divided into two major components: the Programmable Logic (PL) and the Processing System (PS). The PL denotes the FPGA setup and connectivity with the PS. The PS denotes the software that runs on the processors [45]. Based on the work to be completed in this thesis, both the PL and PS are dependent on each other for the UltraZed to function accordingly. The Figure 3.13 depicts how these components interact.

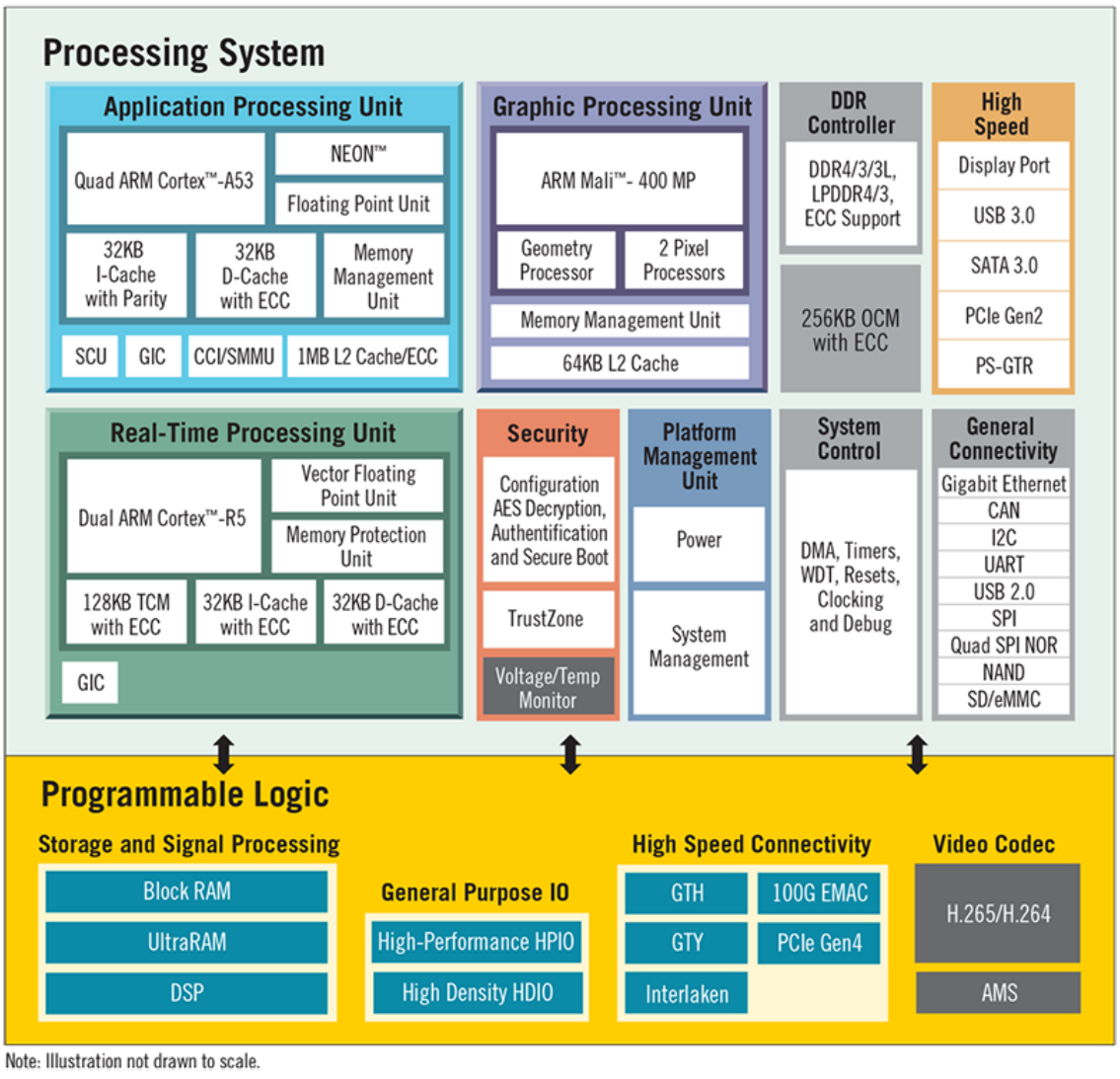


Figure 3.13: Zynq UltraScale+ EV Block Diagram [16]

The Vivado Design Suite [46] is used for the development and setup of the PL in this project. The PS is coded in programming languages like C. It is powered by an Embedded Linux OS created with PetaLinux [47].

3.7.2 Zynq-7000

The ZedBoard equipped with the Zynq-7000 was utilized instead of the UltraZed-EV during development. This is due to COVID-19's influence on electronic supply chains, which has resulted in a longer manufacturing lead time. If an UltraZed-EV board fails, consumers may have to wait multiple months before getting a replacement. This is detailed in further detail in subsection 4.2.1.

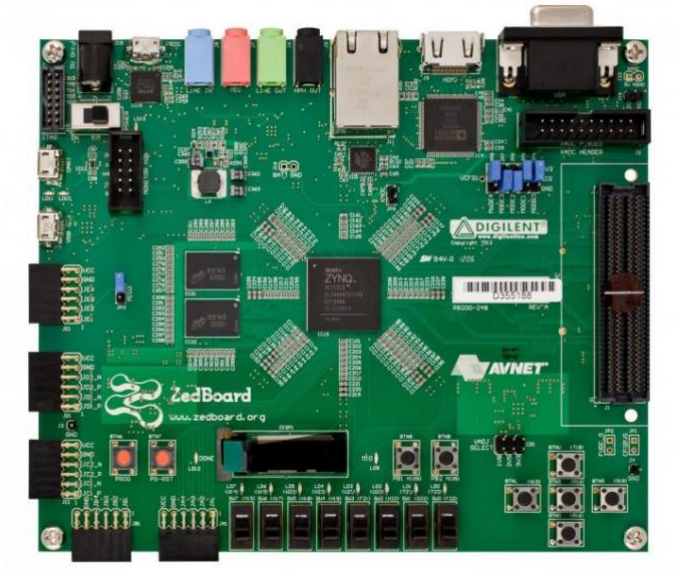


Figure 3.14: ZedBoard Development Kit [17]

Starting up Zynq

The Zynq features on-chip memory (OCM), which is used for booting. The memory is split into two parts: Read-only memory (ROM) and Random-access memory (RAM), having 128 KB and 256 KB of memory, respectively [19]. The steps of the booting procedure are depicted in Figure 3.15.

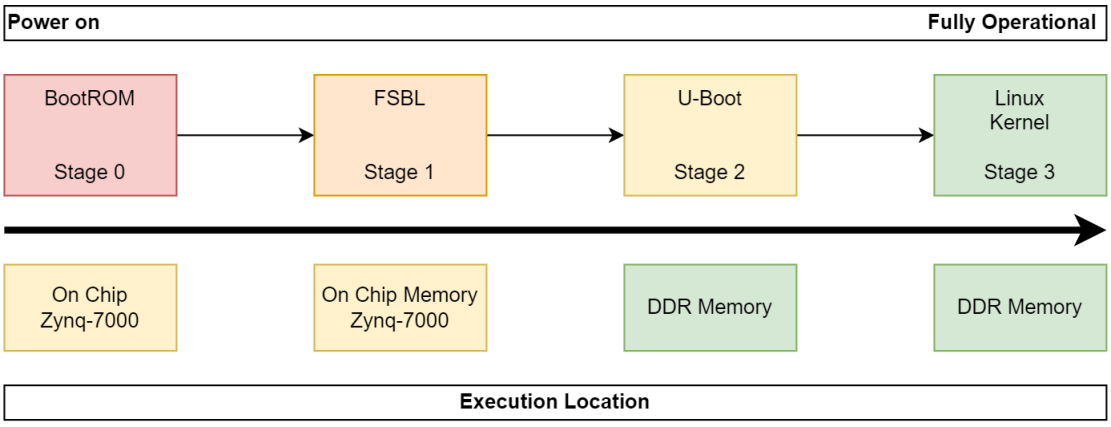


Figure 3.15: Zynq-7000 Boot Sequence [18]

The first stage begins when the chip receives a power-on signal, which occurs when the board's power switch is turned on or when the system reboots. The first thing this step does is read the boot mode signals to determine where the boot source should be loaded from, which can be JTAG, QSPI, flash, or SD card [19].

Xilinx TRM→	MIO[6] Boot_Mode[4]	MIO[5] Boot_Mode[0]	MIO[4] Boot_Mode[2]	MIO[3] Boot_Mode[1]	MIO[2] Boot_Mode[3]
JTAG Mode					
Cascaded JTAG					0
Independent JTAG					1
Boot Devices					
JTAG		0	0	0	
Quad-SPI		1	0	0	
SD Card		1	1	0	
PLL Mode					
PLL Used	0				
PLL Bypassed	1				
Bank Voltages					
MIO Bank 500			3.3V		
MIO Bank 501			1.8V		

Table 3.2: Processing System Boot Mode Selections [25]

Jumpers on pin MIO2 to MIO6 on the board can be used to set different boot mode signals. Once the boot mode has been selected, the boot ROM will read the boot header, verify the image, and load the FSBL image from the selected interface to the OCM using the configuration settings [19].

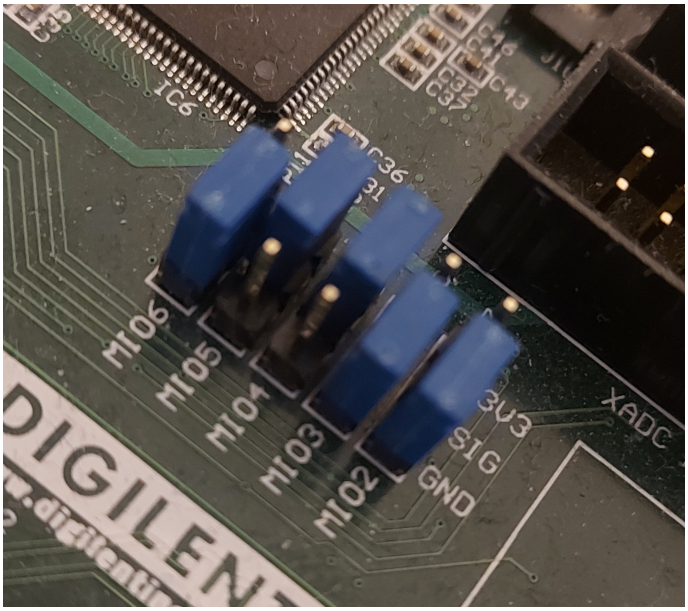


Figure 3.16: SD Card Boot Device Jumper Setting

The function of the First-Stage Boot Loader (FSBL) is the following: load the Linux U-Boot image and execute from non-volatile memory to DDR, initialize the clock phase-locked loop (PLL), initialize the DDR controller, configure the Multiplexed I/O (MIO) and also configures the FPGA with the hardware bitstream (if it exists) [48]. If any user applications are present, the U-boot will run them. The kernel image is then loaded into memory as a compressed image of the Operating System (OS) and decompressed. In addition, the kernel includes a device tree and a ram disk file. The kernel checks the device tree to see what hardware is available, and the ram disk aids in the loading of the root file system [49].

The files necessary for the Linux boot procedure are shown in Figure 3.17. The Zynq boot image file named **BOOT.bin**. It consists of two mandatory files, the FSBL and the SSBL (U-boot) produced by bootgen [48], as well as an optional FPGA bitstream file. The FSBL and SSBL files contain the final stages of the bootloader, which is used to load Linux on the device, as their names imply. The bitstream is the configuration file for the Zynq-7000 AP device's programmable logic. Lastly, the device tree, ramdisk, and compressed Linux kernel are added to an image file known as **image.ub**

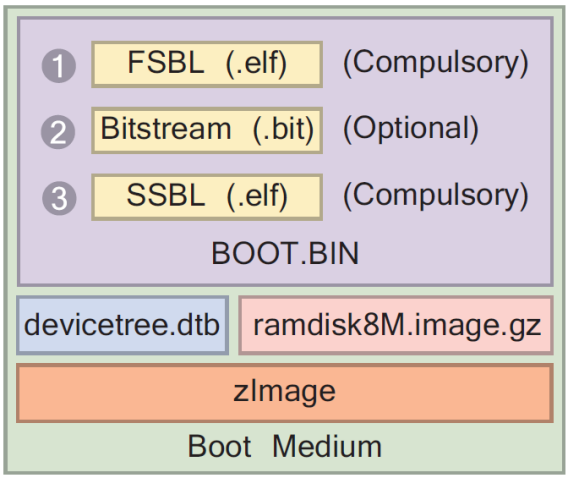


Figure 3.17: Requried files for Zynq Linux boot process [19]

3.7.3 Board Support Package

A domain or board support package (BSP) is a collection of software drivers and, if desired, the operating system on which the application is built. It's the support code for a certain hardware platform or board that aids in basic startup and allows software programs to operate on top of it. One may use the domain to run numerous apps. A domain is associated with a single platform processor [50].

3.8 Tools

Developing embedded systems necessitates the use of software to design and program the various layers of the system. Three separate tools are utilized in this project:

1. Vivado Design Suite (Hardware Development)
2. Vitis (Software Development)
3. PetaLinux (Deploying Embedded Linux)

3.8.1 Vivado Design Suite

Vivado, which was announced in April 2012 [51], is an integrated design environment (IDE) with system-to-IC level capabilities built on a single scalable data model and a common debug environment. Vivado comprises electronic system level (ESL) design tools for synthesizing and testing C-based algorithmic IP; standards-based packaging of both algorithmic and RTL IP for reuse; standards-based IP stitching and system integration of all types of system building blocks; and block and system verification [52].

IP Integrator

The Vivado IP integrator tool allows the user to develop sophisticated system designs on a design canvas by instantiating and interconnecting IP from the Vivado IP catalog. Designs may be created interactively using the IP integrator canvas GUI or automatically via a Tcl programming interface [53].

Tcl Programming Language

Tcl has commands for reading and writing files to the local file system. This allows the user to build folders dynamically, launch FPGA design projects, add files to projects, and perform synthesis and implementation. Tcl can also be used to implement new design techniques or to work around current challenges, such as inserting and removing design objects or changing attributes as needed [54].

Developing Hardware Files

Vivado will produce hardware files after the design is complete. These are two files: a bitstream (.bit) and a hardware design file (.xsa). These two files may then be utilized in software development tools like Vitis and embedded Linux systems like PetaLinux. The bitstream describes an FPGA's logic, whereas the hardware definition file provides information on the various IPs, such as addresses and settings.

3.8.2 Vitis

Vitis is a Xilinx embedded programming IDE that integrates with the open-source software Eclipse [55]. The software provides the user with a high-level coding GUI, allowing them to write in either C or C++ if object-oriented programming (OOP) is preferred. Vitis makes use of hardware files provided by the Vivado Design Suite. Everything needed to design and deploy on a Xilinx board is included in the program.

3.8.3 PetaLinux

PetaLinux is a toolchain that includes everything needed to modify, produce, and deploy Embedded Linux solutions on Xilinx processing systems. It also allows users to customize the boot loader, Linux kernel, or Linux applications. The OPU's operating system is an Embedded Linux OS created with the Xilinx tool Petalinux [47].

3.9 Operating Systems

An operating system is a program that governs the execution of applications and serves as an interface between the computer user and the computer hardware [56].

All of the software and hardware on a computer are managed by the operating system (OS). It handles input and output, as well as regulates peripheral devices.

Several computer applications are usually operating at the same time, and they all require access to the computer's central processing unit (CPU), memory, and storage. All of this is coordinated by the operating system to ensure that each software receives the resources it requires.

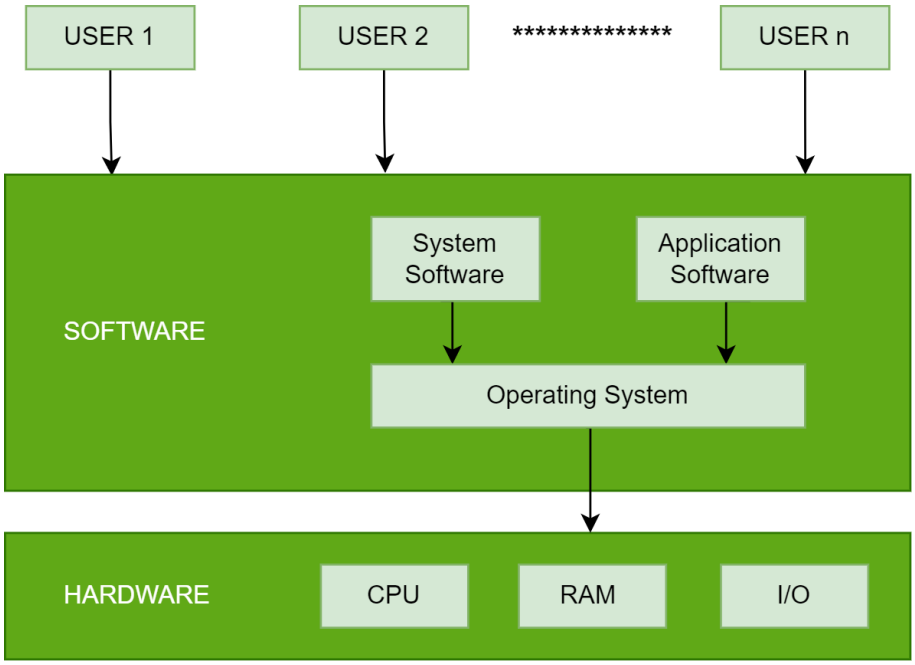


Figure 3.18: Overview of an Operating System

Open source Linux-based operating systems are a common choice for embedded systems. In comparison to commercial operating systems, open-source operating systems are more configurable and adaptable.

The Linux kernel is the core of the Linux operating system. The kernel has access to all of the associated resources and distributes them as the user applications request [43].

3.9.1 Embedded Linux

The OPU's operating system is an Embedded Linux OS created with the Xilinx tool Petalinux as mentioned in subsection 3.8.3. The OS on the OPU is designed to operate on Xilinx processors and perform certain tasks with limited features. It is accessed via a command line, where the user may browse a file system using commands. Memory in Linux is divided into two halves shown in Figure 3.19 user space and kernel space where in user space the user applications are executed and in kernel space kernel code is being executed. Usually, the kernel space has complete access to hardware and system resources, whereas user space has limited access to some of the kernel functions through system calls [49].

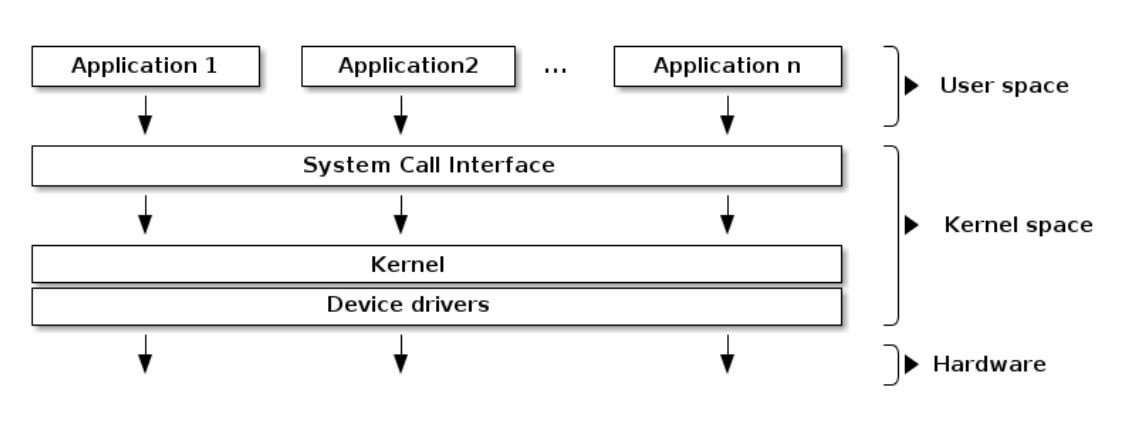


Figure 3.19: Linux Operating System Architecture [20]

Kernel Device Drivers

In Linux, some system functions are only accessed through the kernel space. In order to access these, the Linux developer can use the Kernel Device Drivers (KDD). It is possible to develop a custom kernel driver in the event of missing functionality, which means there is no supported kernel driver for a certain task. These are known as loadable kernel modules (LKMs), and they are written in C-code in the same way as user applications are. The LKM differs from executables in that it utilizes kernel libraries and is built in a different way [43].

Startup and Exit of LKM

Another difference between the LKM and an executable is the method they're developed. A *main()* function is used by an executable to run the program. However, an LKM utilizes two functions: *init* and *exit*. Upon loading the LKM the *init* function starts. Initializing, creating devices, allocating memory, and setting up hardware are all common tasks in this function. Whenever the LKM is unloaded, the exit function is called. Unloading an LKM is uncommon because it disables hardware and removes devices [43].

Virtual Memory

The concept of virtual memory is that the program sees a block of memory of a specific size. This memory can be used by the program in any way it sees fit. The memory block is virtual in the sense that it is made up of many pieces. Some of it may be stored in the computer's main memory, while others may be stored on a disk.

3.9.2 Cube DMA

The Cube DMA core is an FPGA core that specializes in Direct Memory Access (DMA) of cube data, and it employs two standard approaches for arranging picture data for multiband images.

- **BIP** - Band Interleaved by Pixel
- **BSQ** - Band Sequential

The Cube DMA is made to read bit-depths that aren't byte multiples and aren't stored with padding. It is possible to read a cube block-by-block, which implies that if the accelerator requires data to be streamed in blocks, one block is streamed completely before the next. The only restriction is that the block sizes must be multiples of two [21].

As mentioned in the Remote Sensing Techniques section 3.4 Figure 3.26 illustrates the method in which BIP and BSQ cubes are streamed from memory using a four-block cube. The order in which the pixels are read is shown by the red lines. The spatial dimensions of width and height are spatial, whereas the spectral dimension is depth.

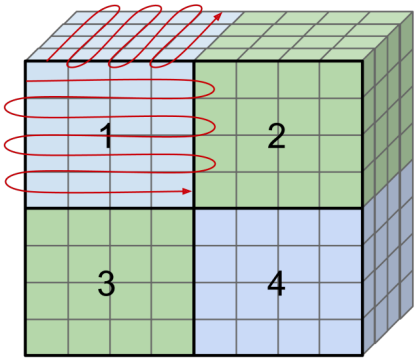


Figure 3.20: BIP order block-wise streaming of HSI cube [21]

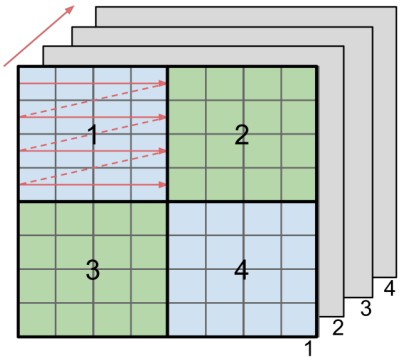


Figure 3.21: BSQ order block-wise streaming of HSI cube [21]

Figure 3.22 shows a block diagram of the Cube DMA. It is made up of two separate channels: Memory Map to Stream (MM2S) and Stream to Memory Map (S2MM). The MM2S gets data from memory and streams it into an accelerator, whereas the S2MM works in the same manner just the opposite. Both channels are configured using a common register interface.

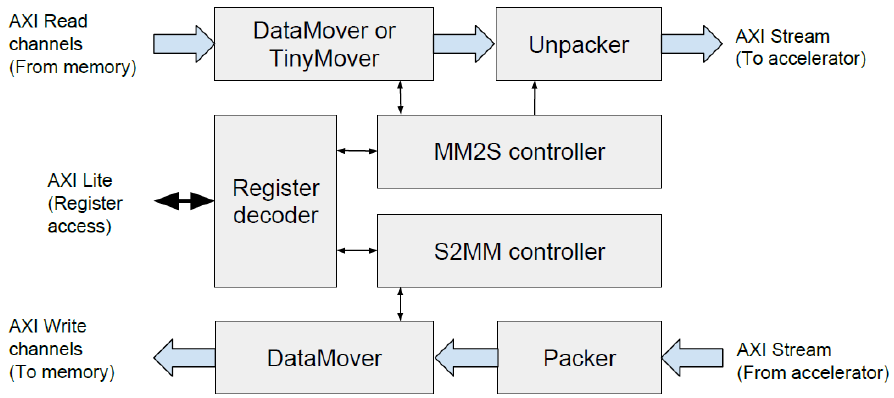


Figure 3.22: Cube DMA Core [21]

The S2MM channel has three components: a packer, a datamover, and a controller. The packer gets data from the accelerator and converts it to the proper word size before sending it to the datamover. The datamover then writes the data to memory in a sequential manner via the AXI bus.
Similarly the MM2S channel also consists of three parts: datamover/tinymover, unpacker, and controller. The datamover/tinymover is in charge of reading from memory directly over the AXI bus. The datamover is a Xilinx IP [57] used for BIP transfers in this architecture, whereas the tinymover is developed by the Cube DMA core designer and used for BSQ transfers.

The unpacker receives the lines fetched from memory through the datamover or tinymover depending on whether it is a BIP- or BSQ transfer. The unpacker aligns the data and adjusts the word sizes. Figure 3.23 depicts the shifting and placement of a 64-bit word of unaligned 12-bit data from memory into four 12-bit sample words. Figure 3.28 shows how a 64-bit word of unaligned 12-bit data from memory is shifted and placed into words containing four 12-bit samples.

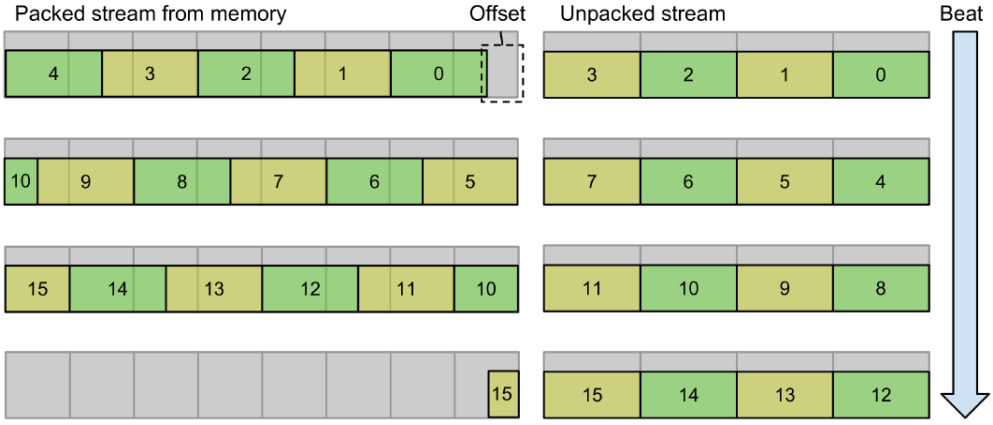


Figure 3.23: Unpacker example [21]

During the transfer, the controller handles the datamover and unpacker, as well as providing a control stream that an accelerator may require. The control stream contains information such as whether a sample is the final pixel in a block, whether it is in the last column, and if it is in the last row [21].

Chapter 4

Implementation

4.1 Overview

There were various working phases involved in completing this project. First, it was necessary to study how the present system operates, particularly the aspects pertaining to data transfer functions. Then it was necessary to identify and design solutions to the technological challenges. Learning a new series of tools, Linux OS, and also programming languages, as well as strengthening partially known programming knowledge, were all part of the process. Interactions with the other developers and participants in the HYPSO project were also required to ensure that the product is properly integrated with the rest of the system.

4.2 System Analysis

This project required a considerable amount of research. The research began with the goal of better understanding the present system and learning how a Zynq-7000 SoC operates. Various tutorials and forums/servers were utilized to master the tools Vivado and Vitis, as well as the hardware design flow. The following web resources were really informative:

- Edaboard.com
- Reconfigurable Embedded Systems with Xilinx Zynq APSoC
- Udemy FPGA Courses

Reading theses from past HYPSO students helped with the comprehension of the OPU system.

4.2.1 Equipment

In addition to the equipment stated in the preceding chapter, the research and prototyping included the use of measurement devices and a prototype board other than the UltraZed board. The following equipment was used:

- **ZedBoard** - Development kit using the Zynq-7000 All Programmable SoC.
- **Analog Discovery 2** - USB oscilloscope, logic analyzer, and multi-function instrument

COVID-19 Impact on Electronics Supply Chain

The worldwide supply chain has been greatly impacted by the COVID-19 outbreak. Multiple nationwide lockdowns continue to stall or even halt the movement of raw materials and completed goods, causing manufacturers to suffer [58].

As a result, it was determined that electronics experiments should be carried out on the ZedBoard rather than the UltraZed. This is due to the fact that SmallSatLab has more ZedBoards than UltraZeds. If something goes wrong with the board and causes it to malfunction, it is preferable that it happen to the ZedBoard rather than the UltraZed. The Zynq-7000 and Zynq UltraScale+ MPSoC families share portions of the same architecture, making design compatibility simple.

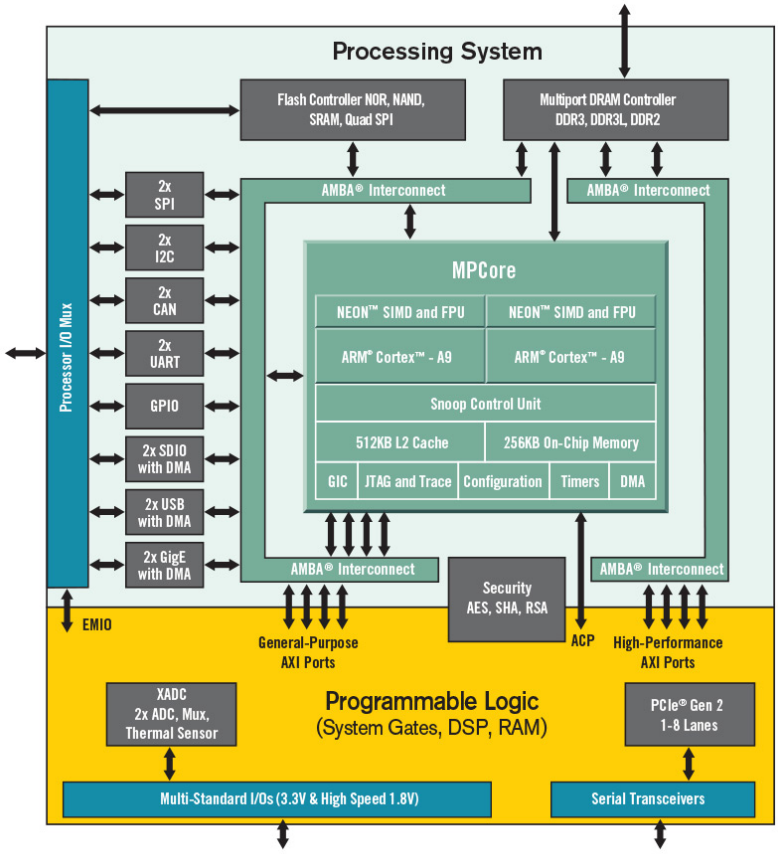


Figure 4.1: Zynq-7000 Block Diagram

4.2.2 AXI UART 16550 v2.0

The AXI UART 16550 core is capable to perform parallel-to-serial conversion on characters received from the AXI master. The AXI UART 16550 is capable of transmitting and receiving 8, 7, 6, or 5-bit characters, with 2 or 1 stop bits and odd, even, or no parity. The AXI UART 16550 can transmit and receive independently, making it full-duplex [22]. The internal registers in the core keep track of its status in the specified state. Receiver, transmitter, and modem control interruptions can all be signaled which is explained in more detail in subsection 4.4.4

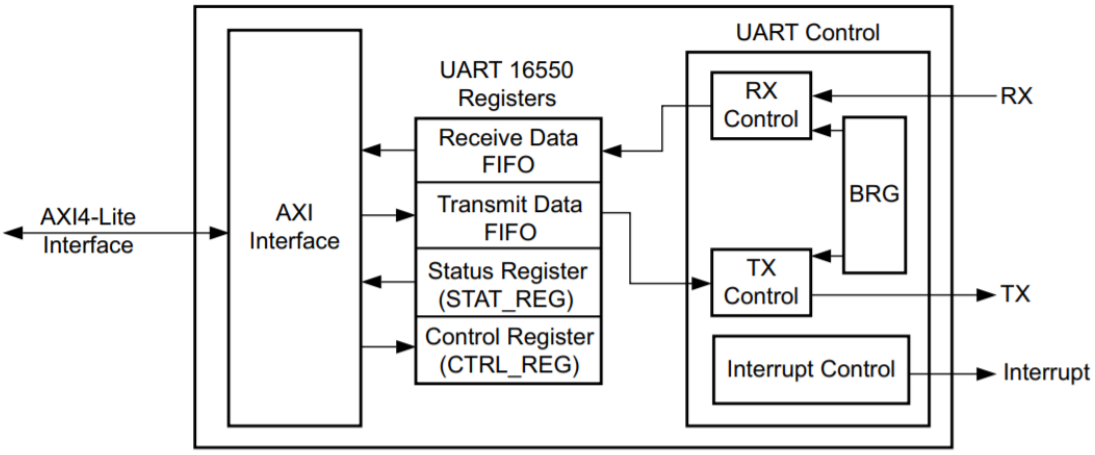


Figure 4.2: The AXI UART 16550 core's top-level block diagram [22]

FIFOs

The First-In, First-Out (FIFO) buffer is intended to increase communication performance; however, if it is configured wrong, communication performance may suffer. The AXI UART16550 is equipped with separate, 16-character-length transmit and receive FIFOs, as shown in Figure 4.2.The software can be used to enable or disable FIFOs which is mentioned further in the paper section regarding the setup of the UART instances 4.4.4.

With a wide receive FIFO, such as the one found in the 16550, the CPU can read all of the data for each receive interrupt at once, saving CPU resources by not having to read data continuously or also known as polling [23].

A Receive Trigger Level (RTL) can be specified in the receive FIFO, indicating that the UART will send an interrupt if the quantity of received data surpasses a certain threshold. The AXI UART 16550 supports the following FIFO trigger levels: 1, 4, 8 and 14 bytes of data [22].

It's worth noting that the highest threshold is 14 bytes rather than the FIFO's maximum capacity of 16 bytes. This is to avoid overflow [23].

More about this functionality, and how it is used is explained in subsection 4.4.4

When selecting a high FIFO threshold, keep in mind the following question: when will the UART produce an interrupt if the quantity of data received hasn't met the RTL? The UART receive timeout becomes important at this point. If the

data received falls short of the RTL, the UART will wait the time required to send four bytes before issuing an interrupt [23].
Consider the scenario below as an illustration of how long this 4 byte may be. The following configuration has been chosen:

- 2.32 Mbps baud rate
- 8 data bits
- No parity bit
- 1 start bit
- 1 stop bit

Each byte requires the transmission of a start bit, eight data bits, and one stop bit, for a total of ten bits, as follows:

$$4\ bytes\ time = 4\ bytes * 10\ bits/byte * \frac{1}{2.32\ Mbps} \approx 17.2\mu s \tag{4.1}$$

The threshold should be set such that these timeout interruptions don't happen frequently, as they cause latency, as shown in the image below.

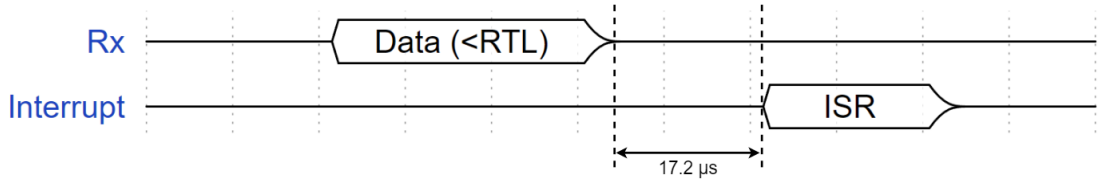


Figure 4.3: Latency Caused by FIFO Timeout [23]

Receiving - Throughput or Latency?

When adding a serial communication protocol to a system, the developer needs to determine whether latency or throughput is more crucial. Keep in mind that the OPU will communicate with the PC via both the CAN and UART protocols, see Figure 1.5. As a result, an RTL value of 8 bytes is preferred, this makes sure to give equal weight to throughput and latency to the receiving end.

4.2.3 Silicon Labs CP2102 USB to UART Bridge

In order to verify that indeed the data does get transmitted between two devices there is to be used a device between the computer and the FPGA to do the translation from the UART signal to USB. For this task, it has been chosen to use the Silicon Labs CP2102 [59].

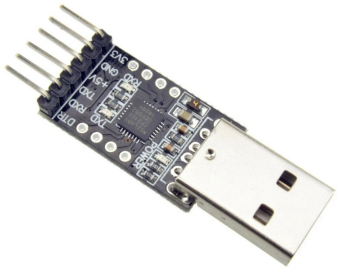


Figure 4.4: CP2102 USB to UART Bridge

4.3 Creating the Base Hardware

In order to be configuring the AXI UART16550 IP to be doing different operations, it needs to have a hardware architecture connecting the programmable logic to the Zynq processing system so that data can be transferred between the sections. The Vivado Design Suite is used to connect the hardware. After the architecture has been realized the bitstream and hardware design file generation will take action and afterward the work is to be done in the software development kit Vitis IDE [60] taking in the usage of the ARM Cortex-A9 CPU [61]

4.3.1 Desired Functionality

It is desired to create a design that will ensure that both data transmission and data reception are tested. For this, two UART modules will be utilized, one for data transmission and the other for data received before transmitting the same data to the computer to test if the received data matches the one sent. For a better understanding, see the diagram below.

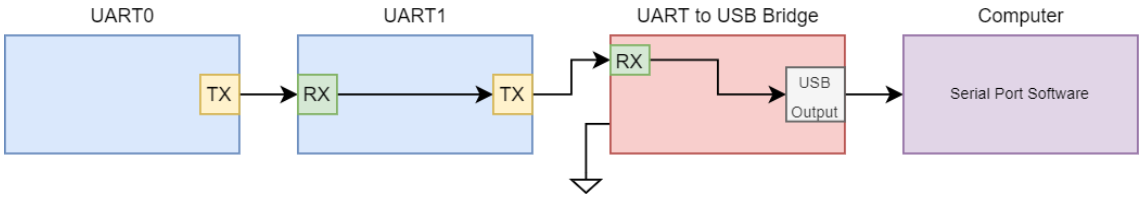


Figure 4.5: Block diagram visualization of the design

ZYNQ7 Processing System IP

The software interface that wraps around the Zynq-7000 Processing System is known as the Zynq-7000 Processing System IP. The Zynq-7000 series is built on a system-on-chip (SoC) integrated processing system (PS) and a Programmable Logic (PL) unit.

The Processing System IP Wrapper connects the PS and the PL logic [62].

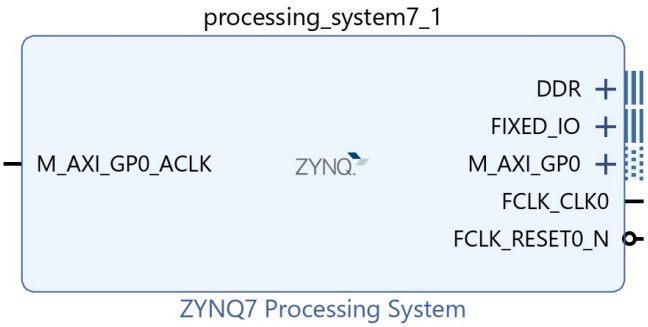


Figure 4.6: ZYNQ7 Processing System

Integrated Logic Analyzer

As mentioned in section 2.1.1 the Integrated Logic Analyzer Figure 4.7 is a logic analyzer core that can be used to monitor a design's internal signals. This IP was very important when prototyping. Being that it allowed to monitor the signals on the Zynq in real-time directly through Vivado. It came to a high value when working with implementing interrupt signals to the UART modules as it made it possible to set triggers for once the interrupt signals occur. Allowing the hardware design to be debugged, see prototyping design Figure 4.10.

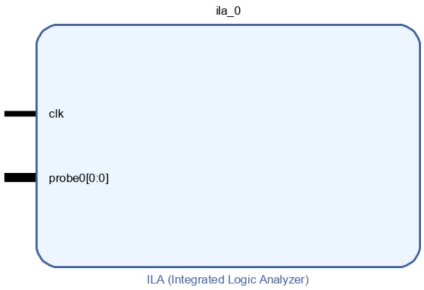


Figure 4.7: Integrated Logic Analyzer IP

AXI UART16550 IP

This IP will take care of the communication to the external world. The IP can take clock signals ranging from 25 MHz to 300 MHz. The higher value of the AXI CLK (Advanced eXtensible Interface Clock) the higher baud rate it is possible to achieve but there is also a drawback of doing so as the energy level will end up increasing. Also, the IP will not be using any external clock, neither for the baud rate nor the external receiver thus both checklists are not checked shown in Figure 4.8. The maximum AXI CLK frequency will be 250 MHz. This is the highest clock frequency (FCLK_CLK0) that the processing system can provide to the programmable logic, see the port to the right in Figure 4.6.

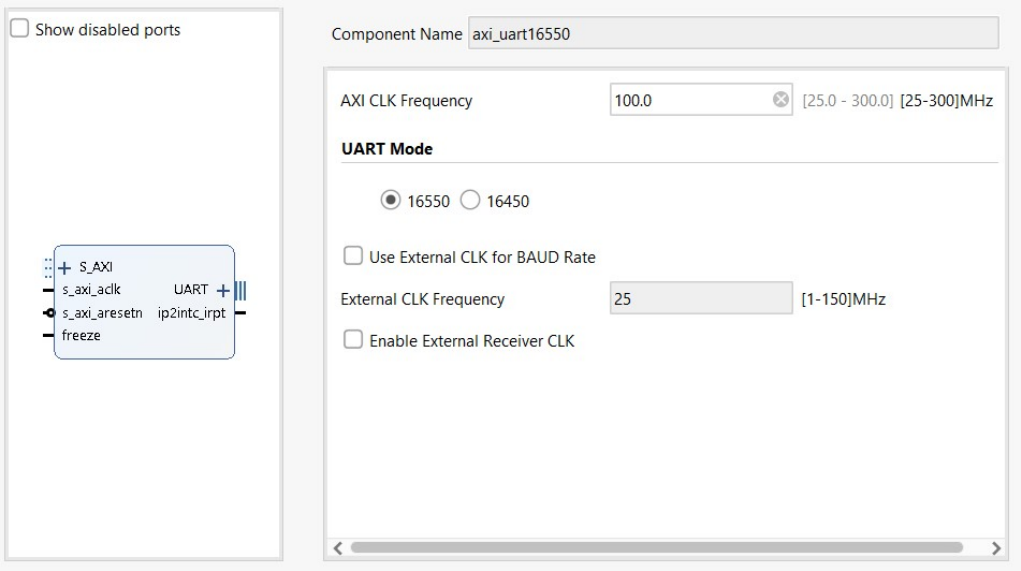


Figure 4.8: AXI UART16550 Configuration

Making Connections

Following the implementation of the ZYNQ7 processing system IP, Integrated Logic Analyzer as well as two AXI UART16550 IP blocks. Block automation is carried out to ensure that the necessary connections are established between the two IPs and the processing system. The transmission UART module is connected to the JA1 peripheral module (PMOD), while the receiver/verification UART is connected to the JB1 PMOD. After completing the block design, the following result is obtained.

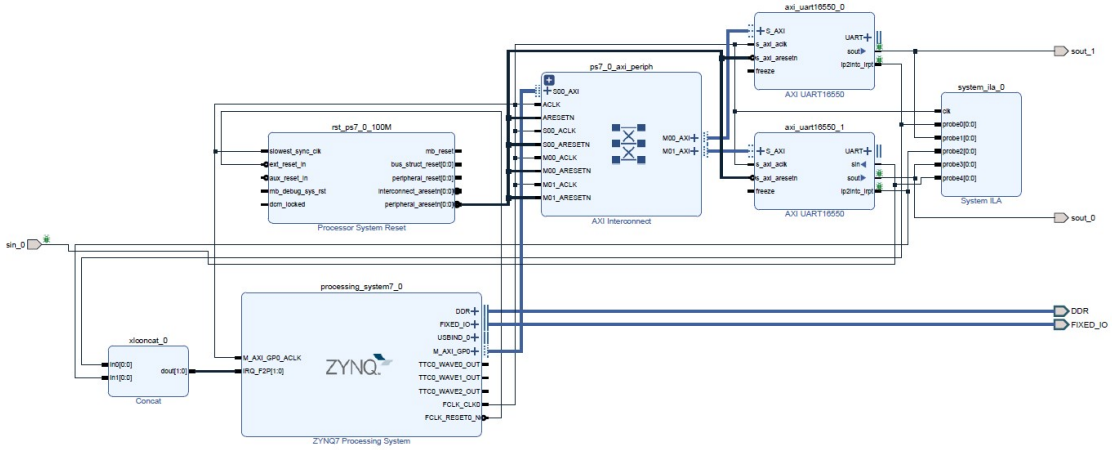


Figure 4.9: AXI UART16550 Prototyping Design

The following Zynq device diagram is obtained after placing and routing the design. The bright orange portion in the top left is the hardened Processing System, which contains the memory controllers and ARM processor cores, while the boxes indicate distinct design sections of the Programmable Logic.

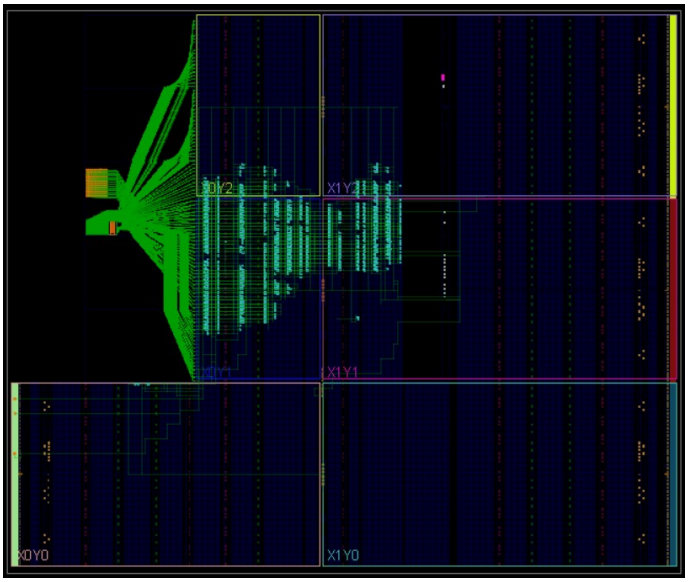


Figure 4.10: The connection between the Processing System and the Programmable Logic on the Zynq-7000 SoC

The links between the PMODs are formed in the following manner:

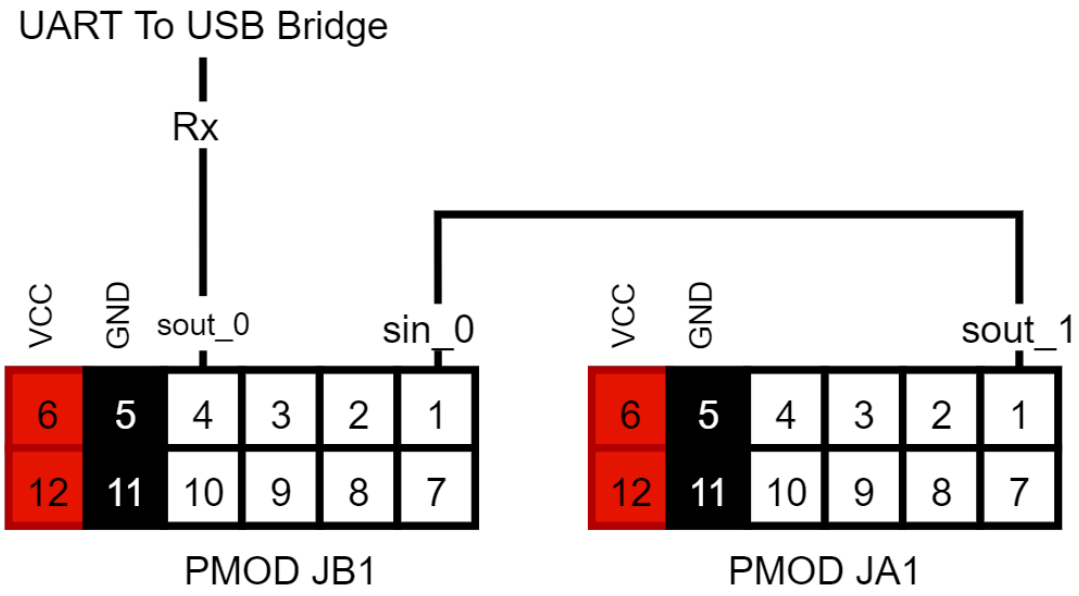


Figure 4.11: The connection between the two UARTs

Again, by performing block automation, both connections between the multiple IPs are formed, and the AXI UART 16550s are automatically allocated memory addresses. The tools assigned addresses to the additional peripherals automatically, making them compatible with the Zynq device. This address is saved in the hardware design file and will be exported to Vitis later. The addresses can be found by opening the *Address Editor* tab as seen in Figure 4.12

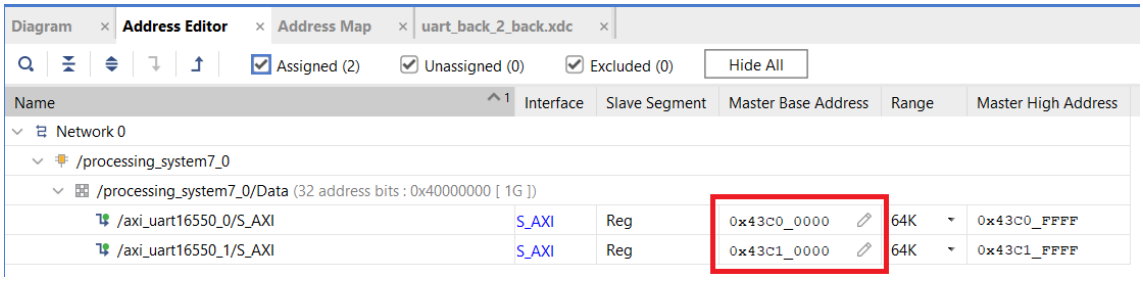


Figure 4.12: Address of AXI UART16550 IP

When dealing with bare-metal programming, Vitis IDE will automatically establish these addresses, but when using PetaLinux, the addresses must be entered in order to be utilized in the source code.

Bitstream Generation and Hardware Export

Following the completion of the preceding procedures, the bitstream will be formed, and the hardware platform will be exported for use in software-related tasks.

4.3.2 Integration

This section will go through how the UART implementation fits into the overall system. This means integrating the AXI UART 16550 into the present system's block architecture and wiring it to it. The modifications to the OPU block design are registered in the ZedBoard's main Tcl (Tool Command Language) script.

Architecture of today's hardware

Obtaining the most recent version of the current block design from the OPU-system repository on GitHub was the first step toward modifying it. The script includes all of the commands required to construct the entire HYPSO OPU system.

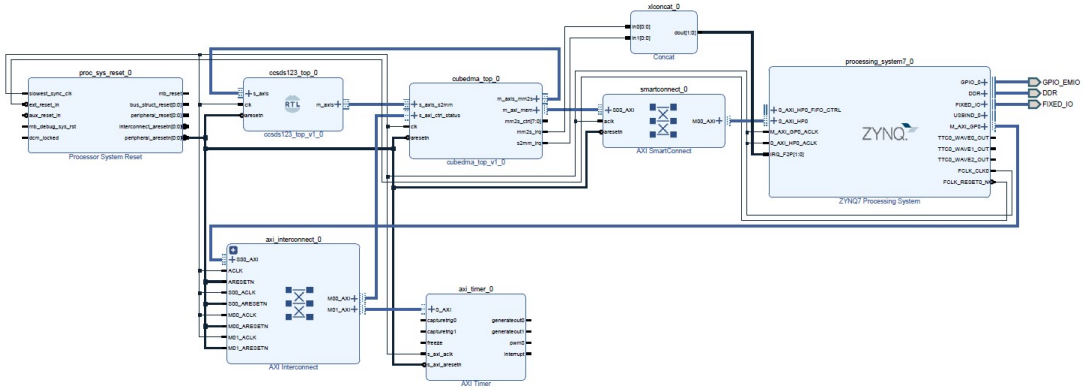


Figure 4.13: Baseline architecture before implementation of AXI UART 16550

The block design before the AXI UART 16550 is implemented is shown in the diagram above Figure 4.13. Several IPs related to compression and cube construction are included in the block design as well as the timestamping functionality.

Integration of the AXI UART 16550

The AXI UART 16550 was added in almost the same way as the programmable logic implementation was described in Section 4.3. The difference in setting up and integrating the AXI UART16550 IP was that connection automation was not being used. Instead, connections between the IP and the rest of the system were done manually to record the Tool Command Language (Tcl) commands that were printed so that they could be added to the system's Tcl-script, automating the OPU design project. The Figure 4.14 shows how the design ended up looking after integrating the AXI UART 16550.

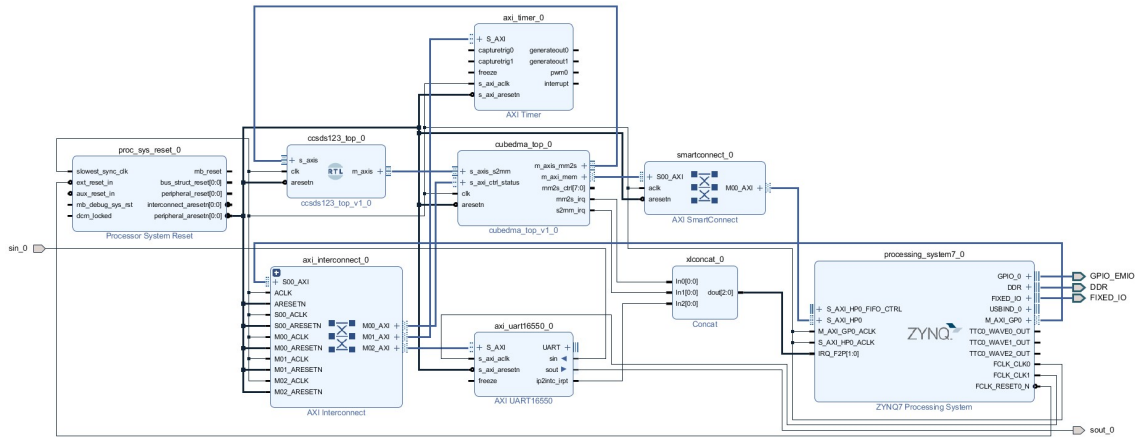


Figure 4.14: Baseline architecture after implementation of AXI UART 16550

TCL Script

Having the hardware architecture design ready for the OPU system, *ZedBoard.tcl* file was to be modified. As mentioned in subsection 4.3.2 the preparation and making of connections between the IPs was done manually and the commands were recorded from the Tcl script seen in Figure 4.15. These commands were saved to a text document which later was added to the main Tcl file.

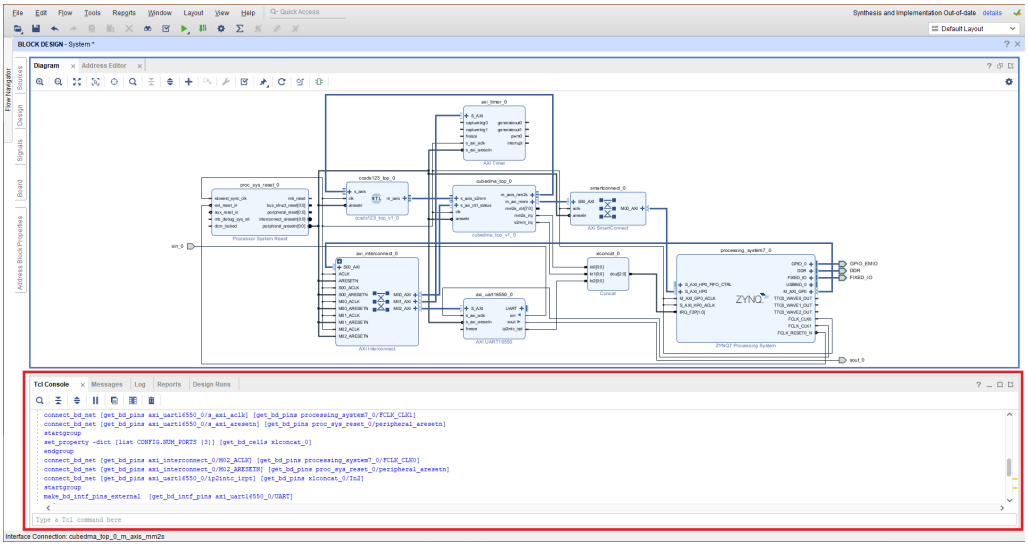


Figure 4.15: Vivado Result Window Area with Tcl Console highlighted

The code below is extracted from the ZedBoard.tcl script and changed for presenting purposes in this report. The full code can be found in Appendix A.

The first modification to the main block design was, of course, the addition of the AXI UART 16550 IP.

```
##################################################################
# Adding the AXI UART16550 block
##################################################################
startgroup
create_bd_cell -type ip -vlnv xilinx.com:ip:axi_uart16550:2.0
↪  axi_uart16550_0
endgroup
```

Since the AXI UART16550 IP is being controlled by the processing system, the AXI Interconnect IP had to add another master interface so that it could connect to the slave interface of the UART IP.

```
##################################################################
# Adding the AXI Interconnect block
##################################################################
startgroup
create_bd_cell -type ip -vlnv xilinx.com:ip:axi_interconnect:2.1
↪  axi_interconnect_0
endgroup
set_property -dict [list CONFIG.NUM_MI {3}] [get_bd_cells
↪  axi_interconnect_0]
```

The requirement for the UART from the HYPSO team is to be able to achieve a baud rate as close to 3 Mbps as possible. Therefore the clock frequency which is provided to the UART IP is chosen as high as the processing system is capable of. Being that the PL Fabric Clock *FCLK_CLK0* is already been set to 100 MHz another component *FCLK_CLK1* has been added to the Zynq Processing System generating a clock frequency of 250 MHz.

```
# Enabling Second Clock for PL
set_property -dict [list CONFIG.PCW_EN_CLK1_PORT {1}] [get_bd_cells
↪  processing_system7_0]
set_property -dict [list CONFIG.PCW_FPGA1_PERIPHERAL_FREQMHZ {250}
↪  CONFIG.PCW_EN_CLK1_PORT {1}] [get_bd_cells processing_system7_0]
```

The AXI UART16550 will also implement interrupt functionality. Therefore the Concat IP needs to have 3 input ports

```
##################################################################
# Adding the concat block
##################################################################
startgroup
create_bd_cell -type ip -vlnv xilinx.com:ip:xlconcat:2.1 xlconcat_0
endgroup
set_property -dict [list CONFIG.NUM_PORTS {3}] [get_bd_cells
↪  xlconcat_0]
```

Lastly, connect the ports of the UART IP to the necessary ports.

```
# For the AXI UART16550
connect_bd_intf_net [get_bd_intf_pins axi_uart16550_0/S_AXI]
↪  -boundary_type upper [get_bd_intf_pins
↪  axi_interconnect_0/M02_AXI]
connect_bd_net [get_bd_pins axi_uart16550_0/s_axi_aclk] [get_bd_pins
↪  processing_system7_0/FCLK_CLK1]
connect_bd_net [get_bd_pins axi_uart16550_0/s_axi_aresetn]
↪  [get_bd_pins proc_sys_reset_0/peripheral_aresetn]
connect_bd_net [get_bd_pins axi_uart16550_0/ip2intc_irpt]
↪  [get_bd_pins xlconcat_0/In2]
connect_bd_net [get_bd_pins axi_interconnect_0/M02_ACLK] [get_bd_pins
↪  processing_system7_0/FCLK_CLK1]
connect_bd_net [get_bd_pins axi_interconnect_0/M02_ARESETN]
↪  [get_bd_pins proc_sys_reset_0/peripheral_aresetn]
```

When assigning the address the range was set to 64K by default. This was reduced to 4K because the UART does not require anything more than that. The offset value was set to 0x43C1 0000, which was also the case with the automated configuration. When accessing the UART registers in the C source code, this address will be utilized.

```
##################################################################
# Set addresses for AXI UART16550
##################################################################
assign_bd_address [get_bd_addr_segs {axi_uart16550_0/S_AXI/Reg }]
set_property range 4K [get_bd_addr_segs
↪  {processing_system7_0/Data/SEG_axi_uart16550_0_Reg}]
set_property offset 0x43C10000 [get_bd_addr_segs
↪  {processing_system7_0/Data/SEG_axi_uart16550_0_Reg}]
```

4.4 Software Development

When it came to programming the Processing System, there were two basic approaches. Xilinx Vitis was used to program the PS at the start of the project. The software developed there is bare-metal, meaning it does not run an operating system. This was utilized in the prototype and implementation of the project. For further information, see section 4.4.1. After being satisfied with the UART functionality it had to become compatible with the rest of the OPU system running Embedded Linux.

4.4.1 Bare-metal configuration

After being pleased with the groundwork in Vivado, the next step is to do embedded C programming in Vitis IDE to make the IP blocks act in accordance with the desired functionality. As previously stated, bare-metal software operates without the use of an operating system. The system will run an application before closing down. The user must relaunch the application when it has completed its execution. A machine with an operating system may run numerous programs, frequently at the same time. Vitis allows users to develop C code and build it for Zynq processors. Vitis downloads hardware data and system settings from Vivado and uses them to create boot files. Boot-files are binary files that are sent to the target device through a cable or an SD card a more detailed description can be found in sub section 3.7.2.

This section will mostly cover the various libraries and drivers that are used, as well as how they work at the code level. Learning how to utilize drivers and comprehend the theory of utilizing the AXI UART 16550 and its interrupts was aided significantly by doing software programming in Vitis IDE.

Baud Rate

The UART features an inbuilt baud rate generator see Figure 4.2 inside the UART control core. This is clocked at the set input clock frequency. Some clock frequencies cannot produce all baud rates, thus it is needed to increase the clock such that higher baud rates can be achieved. The desired baud rate is validated against the allowed error range using the system clock. Some functions may produce an error indicating that the baud rate could not be generated.

Interrupts

An interrupt in embedded processing is a signal that causes the processor's present activity to be briefly halted. To handle the reason for the interrupt, the CPU stores its current state and runs an interrupt service procedure [63]. Interrupts are implemented in the software when a rapid response to an event is required. Especially if the event is asynchronous and the arrival time is unpredictable.

Benefit of An Interrupt Driven Approach

Using interrupts allows the CPU to keep working until an event happens, at which point it can respond to the event. This interrupt-driven strategy also allows for quicker event response times than a polled approach, which involves software actively sampling the state of an external device in a synchronous way.

Interrupt Structure of Zynq SoC

To process interrupts, the Zynq SoC employs a Generic Interrupt Controller (GIC), which can be seen in 4.1 inside the MPCore inside the Processing System and also in the Application Processing Unit (APU) inside the Zynq Ultrascale+ EV 3.13. This is the component in charge of interrupting processing [63]. The GIC may manage interrupts from a variety of sources, including:

- Software-generated interrupts (SGI)
- Shared peripheral interrupts (SPI)
- Private peripheral interrupts (PPI)

SPI will be utilized in the design to implement the AXI UART 16550 IP. Because this form of interrupt comes to/from the device's programmable logic (PL) side. They are shared by the two CPUs of the Zynq SoC. Choosing the trigger type of the Interrupt Request (IRQ) source to be rising edge sensitive.

Processing Interrupt On Zynq SoC

The CPU on the Zynq SoC will do the following steps in response to an interrupt [63]:

1. The interrupt appears to be pending.
2. The processor halts the current thread's execution.
3. The processor saves the thread's status on the stack so that processing may continue after the interrupt has been handled.
4. The interrupt service routine, which specifies how the interrupt should be handled, is executed by the CPU.
5. After recovering the interrupted thread from the stack, the processor continues its activity.

Two functions were used to create the interrupt structure.

- **Interrupt Setup** - Enabling the UART modules to be connected to the interrupt procedure.
- **Interrupt Service Routine (ISR)** - Informs the CPU on how to respond to an interrupt. When the CPU receives an interrupt request from the GIC, it accesses the interrupt vector table to determine which ISR to execute.

4.4.2 Xilinx Libraries

By developing bespoke Xilinx drivers, Xilinx has made board development easier. These drivers are distinguished by the use of a "x" in front of their names. For instance, *xparameters.h*. Because they ease the process of controlling and mixing multiple tasks on the board, these drivers are ideal for prototyping a capability such as data transmission and implementing interrupt functionality to the system.

Library used in Example Design

The platform project is added to Vitis when the hardware design is exported from Vivado. The essential header files for the project may be retrieved by going through the platform project and following the path:

```
/uart_platform/export/uart_platform/sw/uart_platform/standalone_domain/bspinclude/include
```

Another approach to locate the necessary header files is to go to the Xilinx GitHub repository and look in the embeddedsw/drivers directory, where *uartns550*, for example, may be found.

The following libraries are included in the source code.

General Libraries

- **xparameters.h** - The system parameters for the Xilinx device driver environment are contained in this file. It is a system representation since it contains the number of each device in the system as well as the parameters and memory map for each device. This file may be viewed by the user to acquire a summary of the devices in their system as well as device parameters.
- **xil_printf.h** - xil_printf is a lightweight *printf* implementation. It is significantly smaller in size (only 1 kB) [64].
- **stdio.h** - Contains the information needed to include input/output routines in our application. For example, *printf*, *scanf*, and so on.

UART Library

- **xuartns550.h** - Used to control the AXI UART 16550 IP implemented on the FPGA. Including various definitions, structs, and functions to make use of the unit.

Interrupt Libraries

- **xil_exception.h** - Used to handle exceptions, mainly used to handle interrupts and call the ISRs.
- **xscugic.h** - The GIC library is used to link ISRs to IRQs from different modules.

4.4.3 UART550 Hello World

The purpose of this application project was to test the UART module's transmission and receiving capabilities at various baud rates.

Setup

By incorporating the library in the code and creating constants to use with it, the UART drivers were set up. The *xparameters.h* library, which provides the UART driver addresses, was used to define the UART16550_0_ID and UART16550_1_ID. When utilizing the IP's built-in functions, the UARTs will be utilized as a pointer to the instance of the UART controller.

```
#include <xuartns550.h>          // This file holds the drivers for
↪  the configuration and use of the Xilinx 16550 UART.
#include <xparameters.h>          // This file contains the
↪  processor's address space and the device IDs.

// Parameter definitions
#define UART16550_0_ID XPAR_AXI_UART16550_0_DEVICE_ID
#define UART16550_1_ID XPAR_AXI_UART16550_1_DEVICE_ID

// Declarations of the UART0 & UART1 instance structs
XUartNs550 uart0;
XUartNs550 uart1;
```

After creating the macros and adding the relevant header files. Both the transmitter and receiver buffers were declared and defined.

```
// Buffer for transmission of data also receiving data
u8 txHelloWorld[] = "Hello World\n\r";
u8 rxHelloWorld[sizeof(txHelloWorld)];
```

Following that, both UART drivers were initialized using references to the *uart0* and *uart1* structs, as well as their respective unique device IDs. This makes it so that both devices are initialized with the following configurations:

- 19200 bps Baud rate
- 8 data bits
- 1 stop bit
- no parity
- FIFOs are enabled with a receive threshold of 8 bytes

```
// Initialize the UART drivers so that they're ready to use
Status = XUartNs550_Initialize(&uart0, UART16550_0_ID);
if (Status != XST_SUCCESS) {
                print("Initialization of transmission UART
                ↪  failed..\n\r");
                return XST_FAILURE;
        }

Status = XUartNs550_Initialize(&uart1, UART16550_1_ID);
if (Status != XST_SUCCESS) {
                print("Initialization of receiving UART
                ↪  failed..\n\r");
                return XST_FAILURE;
        }
```

It is desired to have the baud rate as quickly as possible such that it does not take long for the operation to complete. Setting the baud rates of both UART devices to the same allows data transmission and reception to operate properly.

```
// Make the baud rate of both UART devices correspond
Status = XUartNs550_SetBaudRate(&uart0, baudValue);
if (Status != XST_SUCCESS) {
                print("Failed to set baud rate of transmitting
                ↪  UART..\n\r");
                return XST_FAILURE;
        }

Status = XUartNs550_SetBaudRate(&uart1, baudValue);
if (Status != XST_SUCCESS) {
                print("Failed to set baud rate of receiving
                ↪  UART..\n\r");
                return XST_FAILURE;
        }
```

Transmission and Reception

Following all of the preparations and device setup. The UARTs are configured to send and receive data. The *XuartNs550_Send* and *XuartNs550_Recv* functions can be found in the *xuartns550.h* header file, which are used to do this. The string "Hello World" is delivered from UART0 to UART1. Whenever data is delivered or received, a print statement is used to verify to the user that the data has been sent correctly.

```
// Send the buffer using the UART0
txDataBytes = XUartNs550_Send(&uart0, txHelloWorld,
↪  sizeof(txHelloWorld));

xil_printf("The number of bytes sent from UART0 is %d\n\r",
↪  txDataBytes);
```

```
sleep(1); // Add 1 second delay to confirm that data has been sent

// Receive data from UART1 and store in buffer
rxDataBytes = XUartNs550_Recv(&uart1, rxHelloWorld,
↪  sizeof(txHelloWorld));

xil_printf("The number of bytes received by UART1 is %d\n\r",
↪  rxDataBytes);
```

Finally, the data received by the UART1 device will be sent to the UART-USB bridge connected to the computer in order to ensure that everything is operating correctly by watching the serial port with a program like TeraTerm.

```
for(int i = 0; i < 5; i++){

        // Sending data which is received from UART0 to the
        ↪  UART-USB bridge connected to the computer
        XUartNs550_Send(&uart1, rxHelloWorld, sizeof(rxHelloWorld));

        // One second pause to make sure that entire data is sent.
        sleep(1);
}
```

4.4.4 Interrupts

Overview

The previous design example does verify that the baud rate change works properly. A design example like this does indeed work, but it forces the processor to focus on the status of the UART which is not very efficent since the processor has to focus on more important tasks.

As a solution to this, interrupt functionality is added to the design allowing the operator to be aware of when data has been sent between the UART, and when data has been received as well as the size of the data in order to see if it corresponds well.

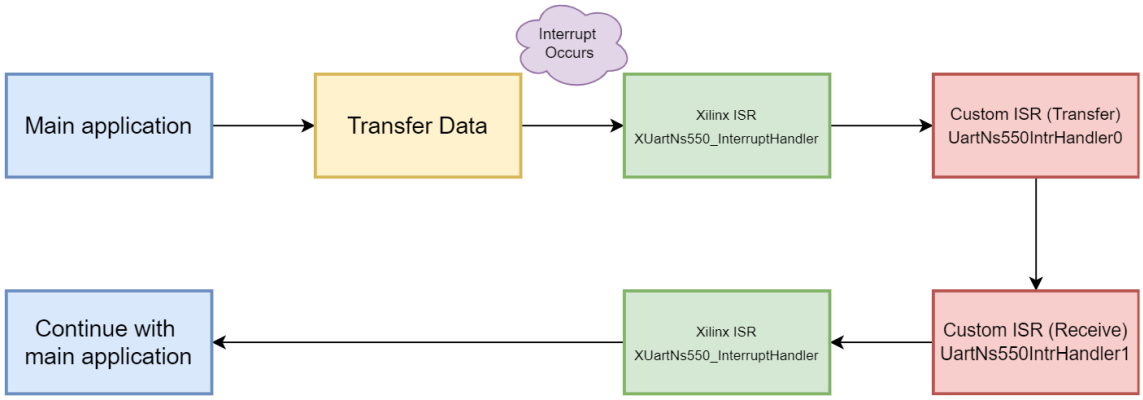


Figure 4.16: Interrupt flow for the design

Setup

The program will mainly be executing as in the previous design. Only thing different is as mentioned once the program transfers data it will jump into it's interrupt handler, also known as interrupt service routine (ISR) notifying when data has been transferred, and when it has been received i.e. Associating the AXI UART ISRs with the UART callback functions. These two custom handlers is set by the UART system itself. The callback will notify the user whenever: data is sent or received, timeout occurs or there exists an error in data transmission, more details in paragraph 4.4.4.

```
// Setup interrupt handlers for the UARTs that will be called once
↪  data is transmitted/received
XUartNs550_SetHandler(&UartNs550Instance0, Uart0SendHandler,
↪  &UartNs550Instance0);
XUartNs550_SetHandler(&UartNs550Instance1, Uart1RecvHandler,
↪  &UartNs550Instance1);
```

The device has no way of turning off the receiver, which means the receive FIFO could contain unwanted data. When the driver is initialized, the FIFOs are not flushed, but there is a function that allows the user to reset them if desired [65]. Down below the function that chooses different options is utilizing the following macros:

- **XUN_OPTION_DATA_INTR** - Enable data interrupts
- **XUN_OPTION_FIFOS_ENABLE** - Enable Tx/Rx FIFOs
- **XUN_OPTION_RESET_TX_FIFO** - Reset the Tx FIFO
- **XUN_OPTION_RESET_RX_FIFO** - Reset the Rx FIFO

The following options is set to both of the UART devices.

```
// Enable data interrupt type, enable both Tx/Rx FIFOs and reset
↪  both FIFOs
Options = XUN_OPTION_DATA_INTR | XUN_OPTION_FIFOS_ENABLE |
↪  XUN_OPTION_RESET_TX_FIFO | XUN_OPTION_RESET_RX_FIFO;
Status = XUartNs550_SetOptions(&UartNs550Instance0, Options);
if (Status != XST_SUCCESS) {
        return XST_FAILURE;
}
Status = XUartNs550_SetOptions(&UartNs550Instance1, Options);
if (Status != XST_SUCCESS) {
        return XST_FAILURE;
}
```

UART Format
In the pre-processor directive the user can decide the configurations set for the UART devices. The configuration parameters are the following 4:

1. **BaudRate** - In its default configuration, the controller supports baud speeds ranging from 2400 to 921600 baud [66]. Although, depending on the system clock frequency, higher and lower speeds may be supported.
2. **DataBits** - Five to eight data bits are supported by the UART port [66].
3. **Parity** - Supports parity of none, even, or odd [66].
4. **StopBits** - Number of stop bits possible is either 1 or 2 [66].

Baud Rate
The Zynq-7000 SoC allows the Programmable Logic side of the design to be operated through a fabric clock (FCLK_CLK0...3) which can generate the frequency ranges from 0.1 to 250 MHz [62]. Choosing 250 MHz as the AXI clock frequency for the UART devices it was possible to achieve a baud rate of 2.32 Mbps. This results in transmission time of "Hello World!" being:

$$Transfer\ Time = \frac{Amount\ of\ Data}{Rate\ of\ Speed} = \frac{104\ bits}{2.32\ Mbps} \approx 44.8\mu s \tag{4.2}$$

After initialising and setting up the interrupts as well as the UARTs the system is ready to do transmit and receive operation similarly to what was shown in sub-chapter 4.4.3

Receive FIFO Trigger Level

The function down below is used to set the receive FIFO trigger level. The receive trigger level determines how many bytes in the receive FIFO are required to initiate a receive data event (interrupt). In the interrupt prototyping the RTL was chosen to be at 8 bytes.

```
// Choose UART1 to have a FIFO Threshold at 8 bytes
Status = XUartNs550_SetFifoThreshold(&UartNs550Instance1,
↪  XUN_FIFO_TRIGGER_08);
if (Status != XST_SUCCESS) {
        return XST_FAILURE;
}
```

Handlers

As mentioned in section 4.4.4 the design utilized two interrupt handlers. One for each UART device with different sets of actions:

Uart0SendHandler

The following interrupt handler will make sure to notify the operator whenever the interrupt signal indicating that the entire data has been sent from UART0.

```
void Uart0SendHandler(void *CallBackRef, u32 Event, unsigned int
↪  EventData)
{
        // All of the data has been sent
        if (Event == XUN_EVENT_SENT_DATA) {
                TotalSentCount = EventData;
                xil_printf("UART0 has sent %d bytes to UART1\n\r",
                ↪  TotalSentCount);
        }
}
```

Uart1RecvHandler

For the receiving UART i.e. UART1 it will make sure to notify the operator whenever the following interrupt signals have been triggered:

1. All data has been received
2. Data which is received is less than the threshold selected
3. Data was received, but with an error

```
void Uart1RecvHandler(void *CallBackRef, u32 Event, unsigned int
↪  EventData)
{
        u8 Errors;
        XUartNs550 *UartNs550Ptr = (XUartNs550 *)CallBackRef;

        // All of the data has been received
        if (Event == XUN_EVENT_RECV_DATA) {
                TotalReceivedCount = EventData;
                xil_printf("UART1 has received %d bytes from
                ↪  UART0\n\r",TotalReceivedCount);
        }

        /*
         * Data was received, but not the expected number of bytes
         */
        if (Event == XUN_EVENT_RECV_TIMEOUT) {
                TotalReceivedCount = EventData;
                print("Data which is received from UART0 is lower
                ↪  than threshold set at 8 byte\n\r");
        }

        // Data was received with an error, keep the data but
        ↪  determine what kind of error occured
        if (Event == XUN_EVENT_RECV_ERROR) {
                TotalReceivedCount = EventData;
                TotalErrorCount++;
                Errors = XUartNs550_GetLastErrors(UartNs550Ptr);

                switch (Errors) {
                case XUN_ERROR_BREAK_MASK:
                        print("ERROR: Break detected!");
                case XUN_ERROR_FRAMING_MASK:
                        print("ERROR: Frame error!");
                case XUN_ERROR_PARITY_MASK:
                        print("ERROR: Parity error!");
                case XUN_ERROR_OVERRUN_MASK:
                        print("ERROR: Overrun error!");
                }
```

```
        }
}
```

A comparison between the sent data and the received data is taken, and the interrupt system for both devices are disabled.

Chapter 5

Testing

5.1 Overview

Testing different changes made to the hardware as well as software was a big part of this project work. The majority of the time, the result was negative, indicating that things did not go as expected. This was, still, a important part of the project's development and comprehension. Failure is an inevitable element of every endeavour.

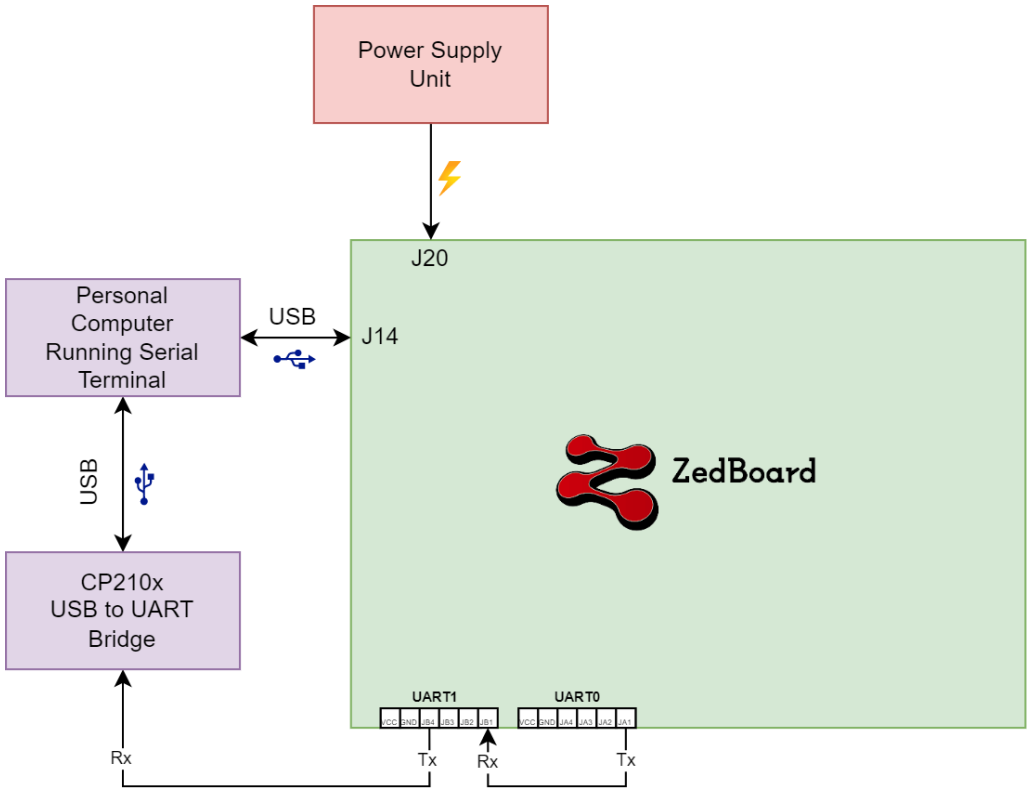


Figure 5.1: Block diagram showing connection with ZedBoard

5.2 Procedure for testing

The board and the wires were connected according to the figure 5.1. The ARM Cortex A9 processor inside of the processing system would through the AXI interface fill the *Transmit Data FIFO* located inside the AXI UART 16550, see Figure 4.2. Having the data now stored into it's register the UART0 transmits the data to the other device UART1 which does store the data into the *Receive Data FIFO* which afterwards is also added to it's own *Transmit Data FIFO* and then sent to the user in order to verify that data was indeed successfully transferred.

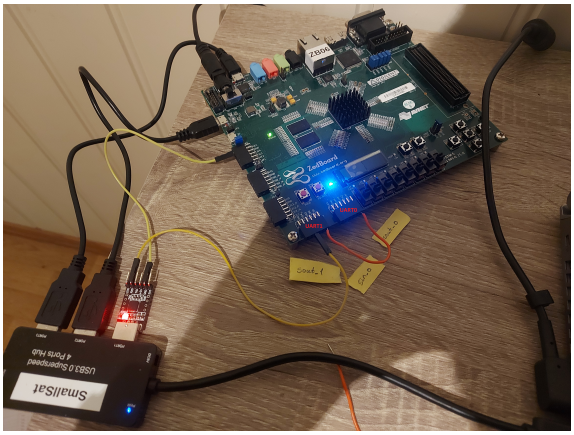


Figure 5.2: Physical connection between UART modules and USB-UART bridge

5.2.1 Verifying Interrupt Service Routines

Similar to the example in subsection 4.4.3, when using the interrupt functionality the data "Hello World!" of size 13 bytes (including null character) is transmitted between the two UARTs.

When data is transferred via UART0, the module's handler notifies the user that data has been sent as well as the amount of bytes sent. In the case of the UART1 module, the user will be notified when all of the data has been received or if a timeout has occurred.

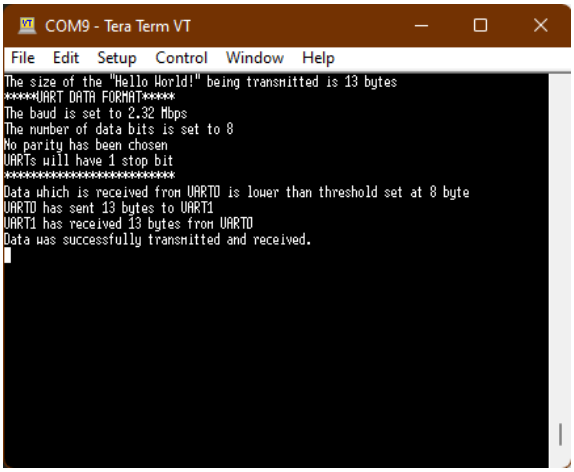


Figure 5.3: Serial Terminal Result after Executing Interrupt Application

When looking at the serial terminal output Figure 5.3 after running the example, it's clear that the interrupt handlers are working appropriately and that data is being sent successfully. However, keep in mind that the receive timeout occurs first. This is due to the fact that the UART1 device starts receiving before the UART0 device begins transmitting since it is loopback connected, see Figure 4.9.

5.2.2 Increasing the baud rate

USB adapters using CP210x chip sets are limited to 32 standard UART baud rates by default. This is acceptable for most purposes, but not if it is desired to use baud rates outside of the standard range [67]. Instead of utilizing the CP210x chip when the baud rate was 2.32 Mbps, the Analog Discovery 2 was chosen instead. Allowing for testing higher baud rates of the UART protocol. The IPs were validated to handle this rate of data transmission/receiving using the Analog Discovery 2's inbuilt digital protocol analyzer.

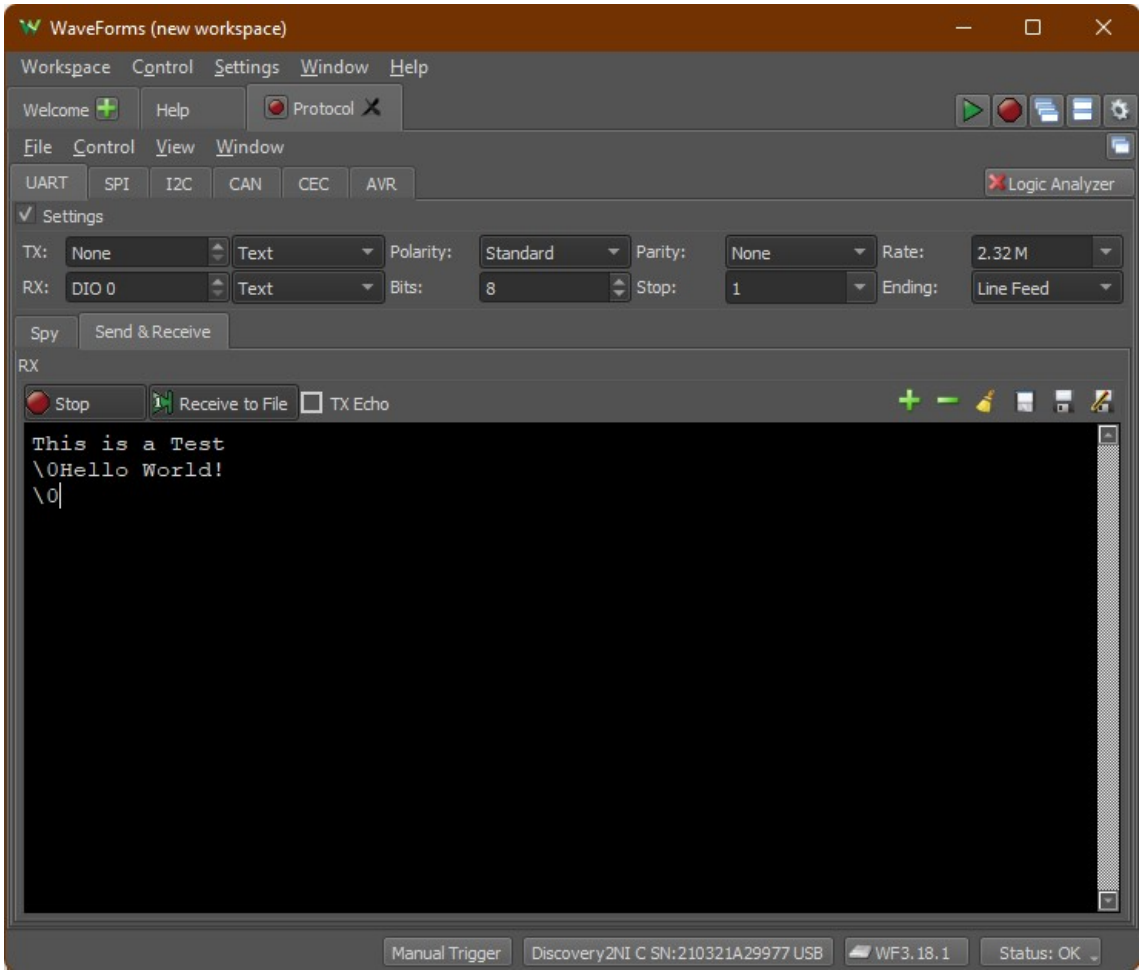


Figure 5.4: Receive result from UART1

Both strings were successfully transmitted and received, as can be seen in the image Figure 5.4 above. Unfortunately, the null character is printed along with the transmitted/received data. By tinkering with the code, this should be easy to resolve.

Chapter 6

Discussion

This chapter will discuss the important aspects of the UART feature. It will cover the choices that were made as well as issues that were raised.

6.1 AXI UART 16550 instead of AXI UART Lite

The Zynq-7000 supports two UART IPs: AXI UART Lite and AXI UART 16550, which was quickly discovered when the project began. Why not just use the UART Lite instead? The reason for this is that the AXI UART 16550 IP has a wider range of options, such as the ability to choose the number of stop bits per UART frame. But most significantly, the gap between the highest baud value for the AXI UART Lite, which is 0.9216 Mbps, and the AXI UART 16550, which was able to attain a baud of 2.32 Mbps, might be much higher by raising the AXI CLK.

6.2 Implementing UART Feature

The UART capability was included in both the hardware design and the bare-metal implementation. It worked as expected, and data was sent between the two devices at a reasonable speed. The third part of the procedure is still to be completed, which is to implement the system in Linux. With the embedded C source code, this should not be a problem, and it will be investigated when the thesis is submitted.

6.3 Coronavirus Disease

The pandemic complicated things further. Overthinking became a big part of the process as a result of being limited on equipment where one defect in the electronics may result in a long wait for a new product, resulting in not allowing myself to explore and allow things to fail. This caused the different processes to be highly precise, but resulted into taking time.

6.4 Development of Hardware

The hardware development part of the logic was split into two parts: prototyping and integrating. This was intended primarily to provide a simpler environment to

work in when understanding the usage of Vivado and the AXI UART 165550. The OPU system involves a significant number of IPs and different configurations. In order to avoid being overwhelmed a separate project was created with only the Zynq PS as well as the UARTs allowing a much easier environment to do prototyping and become familiar with the tool, design flow and the IP. The IP was ready to be integrated with the OPU system once prototyping was completed.

Generating hardware projects demands a significant amount of processing power and might take a long time. When working on the software often times the hardware platform needed to be modified resulting into having to generate bitstream again. Instead of having to generate for the entire system a big amount of time was saved by working in the prototyping environment.

Chapter 7

Conclusion

7.1 Result

After completing this thesis, the end product is an AXI UART 16550 IP that is integrated into the OPU system and is capable of accepting instructions from the PC and transferring data from memory in bare metal, as well as having other features such as different types of interrupt.

The Tcl script, which allows the system to be built automatically, and the embedded C code which helps the team understand how the IP functions, will be of particular interest to the organization. After this thesis is submitted, the Linux operating system utilizing the device will be developed.

7.2 Learning

This project had an exceptionally high learning curve. There was a lot of information on the HYPSO project and the OPU system. To prevent being overworked, the tasks were broken down into smaller chunks.

Working with HYPSO gave me a lot of knowledge. Technical abilities such as creating embedded systems, FPGA and C programming, interrupts, serial protocols and also using various development tools was learnt. Different working methodologies such as planning poker and becoming familiar to status meetings was also a great experience, which will be very useful when starting to work for a company.

Bibliography

[1] NTNU SmallSat Lab, "HYPSO_autod.jpeg (1749×973)."

[2] I. C. Jenssen, "Thermal Analysis on Hyperspectral Imaging Payload for a 6U CubeSat," Sept. 2021.

[3] NTNU SmallSat Lab, "HYPSO-project - 2021 HYPSO-2 satellite architecture.pdf - All Documents."

[4] J. Zabalza, "Feature Extraction and Data Reduction for Hyperspectral Remote Sensing Earth Observation," *University of Strathclyde*, p. 178, June 2015.

[5] NanoAvionics, "6U Nanosatellite Bus M6P."

[6] G. A. Shaw and H.-h. K. Burke, "Spectral Imaging for Remote Sensing," vol. 14, no. 1, p. 26, 2003.

[7] E. Michael, *Hyperspectral Remote Sensing*, vol. PM210. SPIE., Apr. 2012.

[8] F. Dell'Endice, J. Nieke, B. Koetz, M. E. Schaepman, and K. Itten, "Improving radiometry of imaging spectrometers by using programmable spectral regions of interest," *ISPRS Journal of Photogrammetry and Remote Sensing*, vol. 64, pp. 632–639, Nov. 2009.

[9] G. Polder, E. Pekkeriet, and M. Snikkers, *A Spectral Imaging System for Detection of Botrytis in Greenhouses*. July 2013.

[10] J. Fjeldtvedt and M. Orlandić, "CubeDMA – Optimizing three-dimensional DMA transfers for hyperspectral imaging applications," *Microprocessors and Microsystems*, vol. 65, pp. 23–36, Nov. 2018.

[11] Avnet Inc., "UltraZed SOM | UltraZed-EV | Avnet Boards."

[12] M. E. Grøtte, R. Birkeland, E. Honoré-Livermore, S. Bakken, J. L. Garrett, E. F. Prentice, F. Sigernes, M. Orlandić, J. T. Gravdahl, and T. A. Johansen, "Ocean Color Hyperspectral Remote Sensing With High Resolution and Low Latency–The HYPSO-1 CubeSat Mission," *IEEE Transactions on Geoscience and Remote Sensing*, pp. 1–19, 2021. Conference Name: IEEE Transactions on Geoscience and Remote Sensing.

[13] M. Hov, "Design and Implementation of Hardware and Software Interfaces for a Hyperspectral Payload in a Small," 2019. Accepted: 2019-10-31T15:12:35Z Publisher: NTNU.

[14] A. Gjersvik, "Breakout Board Version 3.1 Design Report," p. 48.

[15] FTDI Chip, "What is a UART?," Aug. 2009.

[16] S. Soldavini and A. Ramsey, "Zynq Ultrascale+ Architecture," p. 18.

[17] "ZedBoard | Avnet Boards."

[18] Larry Osborn, "Using U-boot as production test strategy – really?," Nov. 2018.

[19] Louise H. Crockett, Ross A. Elliot, Martin A. Enderwitz, and Robert W. Stewart, *The Zynq Book*. 1st edition ed., July 2014.

[20] The kernel development community, "Introduction — The Linux Kernel documentation."

[21] J. A. Fjeldtvedt, "Efficient Streaming and Compression of Hyperspectral Images," July 2018. Accepted: 2018-10-08T14:00:48Z Publisher: NTNU.

[22] Xilinx Inc., "AXI UART 16550 v2.0 LogiCORE IP Product Guide," Oct. 2016.

[23] Casper Yang, "The Secrets of UART FIFO," Oct. 2009.

[24] Xilinx Inc., "Zynq UltraScale+ MPSoC Data Sheet: Overview (DS891)," p. 42, 2021.

[25] Avnet Inc., "ZedBoard (Zynq™Evaluation and Development) Hardware User's Guide," Jan. 2014.

[26] S. May, "What Is a Satellite?," June 2015. Publisher: Brian Dunbar.

[27] H. Jones, "The Recent Large Reduction in Space Launch Cost," July 2018. Accepted: 2018-07-06T23:25:21Z Publisher: 48th International Conference on Environmental Systems.

[28] A. Nanjangud, P. C. Blacker, S. Bandyopadhyay, and Y. Gao, "Robotics and AI-Enabled On-Orbit Operations With Future Generation of Small Satellites," *Proceedings of the IEEE*, vol. 106, pp. 429–439, Mar. 2018. Conference Name: Proceedings of the IEEE.

[29] "Sentinel-3A," June 2021. Page Version ID: 1031190312.

[30] NTNU SmallSat Lab, "NTNU SmallSat Lab - NTNU."

[31] S. Lee, A. Hutputanasin, A. Toorian, W. Lan, R. Munakata, J. Carnahan, D. Pigatelli, and A. Mehrparvar, "CubeSat Design Specification (CDS) REV 13," p. 42, Feb. 2014.

[32] G. Quintana, R. Birkeland, and E. Honoré-Livermore, "SDR System Design Report," May 2019.

[33] R. Schowengerdt, "Thematic Classification | Elsevier Enhanced Reader," 2007.

[34] Avnet Inc., "PicoZed | Avnet Boards."

[35] Armin Bahadoran, "HYPSO2-RP-003 UART Protocol Design and Functionality," Sept. 2021.

[36] Xilinx Inc., "Virtual Input/Output v3.0," Apr. 2018.

[37] Xilinx Inc., "Integrated Logic Analyzer v6.1," Apr. 2016.

[38] Armin Bahadoran, "Exploration and implementation of communication protocols for satellite payload systems," p. 49, May 2021.

[39] https://github.com/libcsp/libcsp/graphs/contributors, "The Cubesat Space Protocol," Nov. 2021. original-date: 2011-10-07T10:35:34Z.

[40] Magne Hov, "INTEGRATION OF A NETWORK STACK ON A NANOSATELLITE PAYLOAD," Dec. 2018.

[41] G. McNamara, J. Larson, S. Schwartz, and M. Davidson, "Spectral Imaging and Linear Unmixing."

[42] "RS-422," Feb. 2022. Page Version ID: 1073208240.

[43] J. A. Kornberg and S. Netteland, "Time Synchronization of Hyperspectral Image Capture on board a Nanosatellite," p. 203, 2020.

[44] Crystal Chen, Greg Novick, and Kirk Shimano, "RISC vs. CISC," 2000.

[45] Xilinx Inc, "Zynq UltraScale+ MPSoC Processing System v3.3 LogiCORE IP Product Guide," p. 208, 2020.

[46] X. Inc, "Vivado Design Suite Tutorial: Designing IP Subsystems Using IP Integrator (UG995)," p. 40.

[47] Xilinx Inc, "Revision History • PetaLinux Tools Documentation Reference Guide (UG1144) • Reader • Documentation Portal," Nov. 2020.

[48] Xilinx Inc, "ug1165-zynq-embedded-design-tutorial.pdf • Viewer • Documentation Portal," June 2020.

[49] Andreas Varntresk, "Assembly and testing of baseline processing chain," 2019.

[50] Xilinx Inc., "Using the Zynq SoC Processing System," Feb. 2022.

[51] SECURITIES AND EXCHANGE COMMISSION, "XILINX INC (Form Type: 8-K, Filing Date: 04/25/2012)," Apr. 2012.

[52] EDN, "The Vivado Design Suite accelerates programmable systems integration and implementation by up to 4X," June 2012.

[53] Xilinx Inc., "Designing IP Subsystems Using IP Integrator," Oct. 2013.

[54] Xilinx Inc., "Vivado Design Suite User Guide: Using Tcl Scripting (UG894)," p. 114, Dec. 2018.

[55] E. F. Inc, "The Community for Open Innovation and Collaboration | The Eclipse Foundation."

[56] University of Wollongong, "Understanding operating systems - University of Wollongong – UOW."

[57] Xilinx Inc, "AXI DataMover v5.1," Apr. 2017.

[58] Sean Harapko, "How COVID-19 impacted supply chains and what comes next," Feb. 2021.

[59] Silcon Labs Inc., "CP2102 Classic USB to UART Bridge - Silicon Labs," Feb. 2017.

[60] Xilinx Inc., "Vitis Software Platform."

[61] A. Ltd., "Cortex™-A9 Technical Reference Manual," June 2012.

[62] Xilinx Inc., "Zynq-7000 Processing System IP."

[63] A. P. Taylor, "How to Use Interrupts on the Zynq SoC," p. 6, 2014.

[64] Xilinx Inc., "OS and Libraries Document Collection," Oct. 2013.

[65] Xilinx Inc., "uartns550: Main Page," Mar. 2017.

[66] Xilinx Inc, "AXI UART 16550 standalone driver - Xilinx Wiki - Confluence," Nov. 2021.

[67] "How COVID-19 impacted supply chains and what comes next."

Appendix A

ZedBoard Tcl Script

```
##################################################################
# Check if script is running in correct Vivado version.
##################################################################
set scripts_vivado_version 2019.1
set current_vivado_version [version -short]

if { [string first $scripts_vivado_version $current_vivado_version]
 ↪  == -1 } {
   puts ""
   catch {common::send_msg_id "BD_TCL-109" "ERROR" "This script was
    ↪  generated using Vivado <$scripts_vivado_version> and is being
    ↪  run in <$current_vivado_version> of Vivado. Please run the
    ↪  script in Vivado <$scripts_vivado_version> then open the
    ↪  design in Vivado <$current_vivado_version>. Upgrade the design
    ↪  by running \"Tools => Report => Report IP Status...\", then
    ↪  run write_bd_tcl to create an updated script."}

   return 1
}


##################################################################
# Setting paths
##################################################################

# Use this if you are not using docker:
set repoDir
 ↪  C:/Users/*USERNAME*/Code/GitHub/NTNU-SmallSat-Lab/opu-system

# Use this if you are using docker:
#set repoDir /home/hypso

set projectDir $repoDir/vivado/projects

#Setting the name of the project
```

```
set projectName ZedBoard

#*****************************************************************
# Creating the project and specifying the part (xc7z020clg484-1)
#*****************************************************************
create_project $projectName $projectDir/$projectName -part
↪  xc7z020clg484-1 -force


#*****************************************************************
# Using the ZedBoard preset
#*****************************************************************
set_property board_part em.avnet.com:zed:part0:1.4 [current_project]

#*****************************************************************
# Including the CubeDMA project IP repository
#*****************************************************************
set_property  ip_repo_paths  $projectDir/cubedma_7020
↪  [current_project]
update_ip_catalog

#*****************************************************************
# Include the .vhd files from the ccsds123 project
#*****************************************************************
add_files $projectDir/ccsds123/src/predictor.vhd
add_files $projectDir/ccsds123/src/sample_store.vhd
add_files $projectDir/ccsds123/src/weight_update.vhd
add_files $projectDir/ccsds123/src/control.vhd
add_files $projectDir/ccsds123/src/packer.vhd
add_files $projectDir/ccsds123/src/sa_coder.vhd
add_files $projectDir/ccsds123/src/dp_ram_wrapper.vhd
add_files $projectDir/ccsds123/src/dot_pipetree.vhd
add_files $projectDir/ccsds123/src/common.vhd
add_files $projectDir/ccsds123/src/local_diff.vhd
add_files $projectDir/ccsds123/src/local_diff_store.vhd
add_files $projectDir/ccsds123/src/top.vhd
add_files $projectDir/ccsds123/src/pipeline_top.vhd
add_files $projectDir/ccsds123/src/shared_store.vhd
add_files $projectDir/ccsds123/src/dp_ram.vhd
add_files $projectDir/ccsds123/src/residual_mapper.vhd
add_files $projectDir/ccsds123/src/xpm_fifo.vhd
add_files $projectDir/ccsds123/src/fifo.vhd


#*****************************************************************
# Creating block design, and calling it "System"
#*****************************************************************
create_bd_design "System"
```

```
#*****************************************************************
# Including and configuring the CubeDMA module
#*****************************************************************
startgroup
create_bd_cell -type ip -vlnv user.org:user:cubedma_top:1.0
 ↪  cubedma_top_0
endgroup

set_property -dict [ list \
  CONFIG.C_MM2S_AXIS_WIDTH  {64} \
  CONFIG.C_MM2S_COMP_WIDTH  {16} \
  CONFIG.C_MM2S_NUM_COMP    {4} \
  CONFIG.C_S2MM_AXIS_WIDTH  {64} \
  CONFIG.C_S2MM_COMP_WIDTH  {16} \
  CONFIG.C_S2MM_NUM_COMP    {4} \
  CONFIG.C_TINYMOVER        {false} \
] [get_bd_cells cubedma_top_0]


#*****************************************************************
# Including and configuring the ccsds123 module
#*****************************************************************

create_bd_cell -type module -reference ccsds123_top ccsds123_top_0

set_property -dict [ list \
  CONFIG.BUS_WIDTH     {64} \
  CONFIG.COL_ORIENTED  {false} \
  CONFIG.COUNTER_SIZE  {6} \
  CONFIG.D             {16} \
  CONFIG.INITIAL_COUNT {1} \
  CONFIG.ISUNSIGNED    {true} \
  CONFIG.KZ_PRIME      {5} \
  CONFIG.LITTLE_ENDIAN {true} \
  CONFIG.NX            {720} \
  CONFIG.NY            {500} \
  CONFIG.NZ            {107} \
  CONFIG.OMEGA         {13} \
  CONFIG.ON_THE_FLY    {false} \
  CONFIG.P             {3} \
  CONFIG.PIPELINES     {4} \
  CONFIG.R             {64} \
  CONFIG.REDUCED       {false} \
  CONFIG.TINC_LOG      {6} \
  CONFIG.UMAX          {16} \
  CONFIG.V_MAX         {3} \
```

```
  CONFIG.V_MIN         {-1} \
] [get_bd_cells ccsds123_top_0]


#*****************************************************************
# Including and configuring the zynq7 processing system
#*****************************************************************
startgroup
create_bd_cell -type ip -vlnv xilinx.com:ip:processing_system7:5.5
 ↪  processing_system7_0

# Setting up AXI interface
set_property -dict [list \
  CONFIG.preset                          {ZedBoard} \
  CONFIG.PCW_USE_S_AXI_HP0               {1} \
  CONFIG.PCW_USE_FABRIC_INTERRUPT        {1} \
  CONFIG.PCW_IRQ_F2P_INTR                {1} \
  CONFIG.PCW_QSPI_GRP_SINGLE_SS_ENABLE {1} \
] [get_bd_cells processing_system7_0]

# Setting the ENET0 clock to ARM PLL
set_property -dict [list \
  CONFIG.PCW_ENET0_PERIPHERAL_CLKSRC   {ARM PLL} \
]  [get_bd_cells processing_system7_0]

#Configuring CAN0 on MIO 10-11
set_property -dict [list \
  CONFIG.PCW_CAN0_PERIPHERAL_ENABLE {1} \
  CONFIG.PCW_CAN0_CAN0_IO {MIO 10 .. 11} \
]  [get_bd_cells processing_system7_0]


# Configuring memory
set_property -dict [list \
  CONFIG.PCW_UIPARAM_DDR_PARTNO {MT41K128M16 HA-15E} \
  CONFIG.PCW_UIPARAM_DDR_BUS_WIDTH {32 Bit} \
  CONFIG.PCW_UIPARAM_DDR_ECC {Disabled} \
]  [get_bd_cells processing_system7_0]

# Enabling Second Clock for PL
set_property -dict [list CONFIG.PCW_EN_CLK1_PORT {1}] [get_bd_cells
 ↪  processing_system7_0]
set_property -dict [list CONFIG.PCW_FPGA1_PERIPHERAL_FREQMHZ {250}
 ↪  CONFIG.PCW_EN_CLK1_PORT {1}] [get_bd_cells processing_system7_0]

#*****************************************************************
# Adding the concat block
```

```
#****************************************************************
startgroup
create_bd_cell -type ip -vlnv xilinx.com:ip:xlconcat:2.1 xlconcat_0
endgroup
set_property -dict [list CONFIG.NUM_PORTS {3}] [get_bd_cells
↪  xlconcat_0]

#****************************************************************
# Adding the AXI SmartConnect block
#****************************************************************
startgroup
create_bd_cell -type ip -vlnv xilinx.com:ip:smartconnect:1.0
↪  smartconnect_0
endgroup
set_property -dict [list CONFIG.NUM_SI {1}] [get_bd_cells
↪  smartconnect_0]

#****************************************************************
# Adding the AXI Interconnect block
#****************************************************************
startgroup
create_bd_cell -type ip -vlnv xilinx.com:ip:axi_interconnect:2.1
↪  axi_interconnect_0
endgroup
set_property -dict [list CONFIG.NUM_MI {3}] [get_bd_cells
↪  axi_interconnect_0]

#****************************************************************
# Adding the Processing System Reset block
#****************************************************************
startgroup
create_bd_cell -type ip -vlnv xilinx.com:ip:proc_sys_reset:5.0
↪  proc_sys_reset_0
endgroup

#****************************************************************
# Adding the AXI Timer block
#****************************************************************
startgroup
create_bd_cell -type ip -vlnv xilinx.com:ip:axi_timer:2.0 axi_timer_0
endgroup
#set_property -dict [list CONFIG.mode_64bit {1}] [get_bd_cells
↪  axi_timer_0]
#No need for 64bit cascade timer for current timestamp method

#****************************************************************
# Adding the AXI UART16550 block
#****************************************************************
```

```
startgroup
create_bd_cell -type ip -vlnv xilinx.com:ip:axi_uart16550:2.0
 ↪  axi_uart16550_0
endgroup
#****************************************************************
# Running block automation
#****************************************************************
startgroup
apply_bd_automation -rule xilinx.com:bd_rule:processing_system7
 ↪  -config { \
  make_external "FIXED_IO,  DDR" apply_board_preset "1" Master
   ↪  "Disable" Slave "Disable" \
}  [get_bd_cells processing_system7_0]

endgroup

#****************************************************************
# Making connections
#****************************************************************
# For the Zynq7 processing system:
connect_bd_net      [get_bd_pins processing_system7_0/M_AXI_GP0_ACLK]
 ↪  [get_bd_pins processing_system7_0/S_AXI_HP0_ACLK]
connect_bd_net      [get_bd_pins processing_system7_0/IRQ_F2P]
 ↪  [get_bd_pins xlconcat_0/dout]
connect_bd_net      [get_bd_pins processing_system7_0/FCLK_CLK0]
 ↪  [get_bd_pins processing_system7_0/M_AXI_GP0_ACLK]
connect_bd_net      [get_bd_pins processing_system7_0/FCLK_RESET0_N]
 ↪  [get_bd_pins proc_sys_reset_0/ext_reset_in]
connect_bd_intf_net [get_bd_intf_pins processing_system7_0/M_AXI_GP0]
 ↪  -boundary_type upper [get_bd_intf_pins
 ↪  axi_interconnect_0/S00_AXI]
connect_bd_intf_net [get_bd_intf_pins processing_system7_0/S_AXI_HP0]
 ↪  [get_bd_intf_pins smartconnect_0/M00_AXI]

# For the CubeDMA:
connect_bd_net      [get_bd_pins cubedma_top_0/mm2s_irq]
 ↪  [get_bd_pins xlconcat_0/In0]
connect_bd_net      [get_bd_pins cubedma_top_0/s2mm_irq]
 ↪  [get_bd_pins xlconcat_0/In1]
connect_bd_net      [get_bd_pins cubedma_top_0/clk]
 ↪  [get_bd_pins processing_system7_0/FCLK_CLK0]
connect_bd_net      [get_bd_pins cubedma_top_0/aresetn]
 ↪  [get_bd_pins axi_interconnect_0/M00_ARESETN]
connect_bd_intf_net [get_bd_intf_pins
 ↪  cubedma_top_0/s_axi_ctrl_status]  -boundary_type upper
 ↪  [get_bd_intf_pins axi_interconnect_0/M00_AXI]
connect_bd_intf_net [get_bd_intf_pins cubedma_top_0/m_axis_mm2s]
 ↪  [get_bd_intf_pins ccsds123_top_0/s_axis]
```

```
connect_bd_intf_net [get_bd_intf_pins cubedma_top_0/s_axis_s2mm]
↪  [get_bd_intf_pins ccsds123_top_0/m_axis]
connect_bd_intf_net [get_bd_intf_pins cubedma_top_0/m_axi_mem]
↪  [get_bd_intf_pins smartconnect_0/S00_AXI]


connect_bd_net      [get_bd_pins ccsds123_top_0/clk]
↪  [get_bd_pins processing_system7_0/FCLK_CLK0]
connect_bd_net      [get_bd_pins ccsds123_top_0/aresetn]
↪  [get_bd_pins cubedma_top_0/aresetn]


# For the Axi Interconnect:
connect_bd_net      [get_bd_pins axi_interconnect_0/ACLK]
↪  [get_bd_pins axi_interconnect_0/S00_ACLK] -boundary_type upper
connect_bd_net      [get_bd_pins axi_interconnect_0/ACLK]
↪  [get_bd_pins processing_system7_0/FCLK_CLK0]
connect_bd_net      [get_bd_pins axi_interconnect_0/S00_ACLK]
↪  [get_bd_pins axi_interconnect_0/M00_ACLK] -boundary_type upper
connect_bd_net      [get_bd_pins axi_interconnect_0/ARESETN]
↪  [get_bd_pins axi_interconnect_0/S00_ARESETN] -boundary_type upper
connect_bd_net      [get_bd_pins axi_interconnect_0/ARESETN]
↪  [get_bd_pins axi_interconnect_0/M00_ARESETN] -boundary_type upper

# For the Axi SmartConnect:
connect_bd_net      [get_bd_pins smartconnect_0/aclk]
↪  [get_bd_pins processing_system7_0/FCLK_CLK0]
connect_bd_net      [get_bd_pins smartconnect_0/aresetn]
↪  [get_bd_pins ccsds123_top_0/aresetn]

# For the Processor System Reset:
connect_bd_net      [get_bd_pins proc_sys_reset_0/slowest_sync_clk]
↪  [get_bd_pins axi_interconnect_0/M00_ACLK]
connect_bd_net      [get_bd_pins proc_sys_reset_0/peripheral_aresetn]
↪  [get_bd_pins axi_interconnect_0/ARESETN]

# For the AXI Timer:
connect_bd_intf_net [get_bd_intf_pins axi_timer_0/S_AXI]
↪  -boundary_type upper   [get_bd_intf_pins
↪  axi_interconnect_0/M01_AXI]
connect_bd_net      [get_bd_pins axi_timer_0/s_axi_aresetn]
↪  [get_bd_pins proc_sys_reset_0/peripheral_aresetn]
connect_bd_net      [get_bd_pins axi_timer_0/s_axi_aclk]
↪  [get_bd_pins processing_system7_0/FCLK_CLK0]
connect_bd_net      [get_bd_pins axi_interconnect_0/M01_ACLK]
↪  [get_bd_pins processing_system7_0/FCLK_CLK0]
connect_bd_net      [get_bd_pins axi_interconnect_0/M01_ARESETN]
↪  [get_bd_pins proc_sys_reset_0/peripheral_aresetn]
```

```
# For the AXI UART16550
connect_bd_intf_net [get_bd_intf_pins axi_uart16550_0/S_AXI]
 ↪  -boundary_type upper [get_bd_intf_pins
 ↪  axi_interconnect_0/M02_AXI]
connect_bd_net [get_bd_pins axi_uart16550_0/s_axi_aclk] [get_bd_pins
 ↪  processing_system7_0/FCLK_CLK1]
connect_bd_net [get_bd_pins axi_uart16550_0/s_axi_aresetn]
 ↪  [get_bd_pins proc_sys_reset_0/peripheral_aresetn]
connect_bd_net [get_bd_pins axi_uart16550_0/ip2intc_irpt]
 ↪  [get_bd_pins xlconcat_0/In2]
connect_bd_net [get_bd_pins axi_interconnect_0/M02_ACLK] [get_bd_pins
 ↪  processing_system7_0/FCLK_CLK1]
connect_bd_net [get_bd_pins axi_interconnect_0/M02_ARESETN]
 ↪  [get_bd_pins proc_sys_reset_0/peripheral_aresetn]


startgroup
set_property -dict [list \
    CONFIG.PCW_QSPI_GRP_SINGLE_SS_ENABLE {1} \
    CONFIG.PCW_GPIO_EMIO_GPIO_ENABLE {1} \
    CONFIG.PCW_GPIO_EMIO_GPIO_IO {8} \
    ] [get_bd_cells processing_system7_0]
endgroup

startgroup
make_bd_intf_pins_external  [get_bd_intf_pins
 ↪  processing_system7_0/GPIO_0]
endgroup

# More intuitive name
set_property name GPIO_EMIO [get_bd_intf_ports GPIO_0_0]

# Import zedboard_constraints.xdc file
add_files -fileset constrs_1 -norecurse
 ↪  $projectDir/constraints/zedboard_constraints.xdc
import_files -fileset constrs_1
 ↪  $projectDir/constraints/zedboard_constraints.xdc

# Set 'zedboard_constraints' file properties
set file_obj [get_files -of_objects [get_filesets constrs_1] [list
 ↪  "zedboard_constraints.xdc"]]
set_property "file_type" "XDC" $file_obj
set_property "is_enabled" "1" $file_obj
set_property "is_global_include" "0" $file_obj
set_property "library" "xil_defaultlib" $file_obj
set_property "path_mode" "RelativeFirst" $file_obj
set_property "processing_order" "NORMAL" $file_obj
```

```
set_property "scoped_to_cells" "" $file_obj
set_property "scoped_to_ref" "" $file_obj
set_property "used_in" "synthesis implementation" $file_obj
set_property "used_in_implementation" "1" $file_obj
set_property "used_in_synthesis" "1" $file_obj

# Set 'constrs_1' fileset properties
set obj [get_filesets constrs_1]
set_property "name" "constrs_1" $obj
set_property "target_constrs_file" "" $obj


# Regenerate layout (only aesthetics)
regenerate_bd_layout

#****************************************************************
# Save the block diagram
#****************************************************************
save_bd_design

#****************************************************************
# Save the block diagram
#****************************************************************
save_bd_design


#****************************************************************
# Set addressses for cube DMA
#****************************************************************
assign_bd_address       [get_bd_addr_segs
↪  {cubedma_top_0/s_axi_ctrl_status/reg0 }]
set_property range 64K  [get_bd_addr_segs
↪  {processing_system7_0/Data/SEG_cubedma_top_0_reg0}]
set_property offset 0x43C00000 [get_bd_addr_segs
↪  {processing_system7_0/Data/SEG_cubedma_top_0_reg0}]

assign_bd_address       [get_bd_addr_segs
↪  {processing_system7_0/S_AXI_HP0/HP0_DDR_LOWOCM }]
set_property range 512M [get_bd_addr_segs
↪  {cubedma_top_0/m_axi_mem/SEG_processing_system7_0_HP0_DDR_LOWOCM}]


#****************************************************************
# Set addresses for AXI Timer
#****************************************************************
assign_bd_address   [get_bd_addr_segs {axi_timer_0/S_AXI/Reg }]
set_property range 4K [get_bd_addr_segs
↪  {processing_system7_0/Data/SEG_axi_timer_0_Reg}]
```

```
set_property offset 0x42800000 [get_bd_addr_segs
 ↪  {processing_system7_0/Data/SEG_axi_timer_0_Reg}]

#****************************************************************
# Set addresses for AXI UART16550
#****************************************************************
assign_bd_address [get_bd_addr_segs {axi_uart16550_0/S_AXI/Reg }]
set_property range 4K [get_bd_addr_segs
 ↪  {processing_system7_0/Data/SEG_axi_uart16550_0_Reg}]
set_property offset 0x43C10000 [get_bd_addr_segs
 ↪  {processing_system7_0/Data/SEG_axi_uart16550_0_Reg}]

#****************************************************************
# Creating HDL wrapper
#****************************************************************
make_wrapper -files [get_files
 ↪  $projectDir/$projectName/$projectName.srcs/sources_1/bd/System/System.bd]
 ↪  -top
add_files -norecurse
 ↪  $projectDir/$projectName/$projectName.srcs/sources_1/bd/System/hdl/System_wrappe

#****************************************************************
# Set the System_wrapper as top
#****************************************************************
set_property top System_wrapper [current_fileset]

#****************************************************************
# Generating bitstream
#****************************************************************

launch_runs impl_1 -to_step write_bitstream -jobs 8
wait_on_run impl_1
open_run impl_1

#****************************************************************
# Exporting hardware files
#****************************************************************

# Exporting hardware definition file
file mkdir $projectDir/$projectName/$projectName.sdk
file copy -force
 ↪  $projectDir/$projectName/$projectName.runs/impl_1/System_wrapper.sysdef
 ↪  \
                $projectDir/$projectName/$projectName.sdk/system.hdf

# Launch SDK (Doesn't work for docker...)
launch_sdk -workspace $projectDir/$projectName/$projectName.sdk \
```

```
            -hwspec
             ↪  $projectDir/$projectName/$projectName.sdk/System_wrapper.hdf

# Export bitstream w/o SDK
file copy -force
 ↪  $projectDir/$projectName/$projectName.runs/impl_1/System_wrapper.bit
 ↪  \
          $projectDir/$projectName/$projectName.sdk/bitstream.bit

puts "The script finished"
```

Appendix B

Bare-Metal Source Code

B.1 Simple Transmit/Receive Design

```
#include <stdio.h>                          // Standard-input/output
↪  library
#include <xuartns550.h>          // This file holds the drivers for
↪  the configuration and use of the Xilinx 16550 UART.
#include <xparameters.h>          // This file contains the
↪  processor's address space and the device IDs.

// Parameter definitions
#define UART16550_0_ID XPAR_AXI_UART16550_0_DEVICE_ID
#define UART16550_1_ID XPAR_AXI_UART16550_1_DEVICE_ID

// Declarations of the UART0 & UART1 instance structs
XUartNs550 uart0;
XUartNs550 uart1;

int main(){

        int Status;
        uint txDataBytes;
        uint rxDataBytes;

        // Baud rate adjustment
        u32 baudValue = 9600;

        // Buffer for transmission of data also receiving data
        u8 txHelloWorld[] = "Hello World\n\r";
        u8 rxHelloWorld[sizeof(txHelloWorld)];

        // Initialize the UART drivers so that they're ready to use
        Status = XUartNs550_Initialize(&uart0, UART16550_0_ID);
        if (Status != XST_SUCCESS) {
                        print("Initialization of transmission UART
                        ↪  failed..\n\r");
```

```
                return XST_FAILURE;
        }

    Status = XUartNs550_Initialize(&uart1, UART16550_1_ID);
    if (Status != XST_SUCCESS) {
                print("Initialization of receiving UART
                ↪  failed..\n\r");
                return XST_FAILURE;
        }

    // Make the baud rate of both UART devices correspond
    Status = XUartNs550_SetBaudRate(&uart0, baudValue);
    if (Status != XST_SUCCESS) {
                print("Failed to set baud rate of
                ↪  transmitting UART..\n\r");
                return XST_FAILURE;
        }

    Status = XUartNs550_SetBaudRate(&uart1, baudValue);
    if (Status != XST_SUCCESS) {
                print("Failed to set baud rate of receiving
                ↪  UART..\n\r");
                return XST_FAILURE;
        }

    // Send the buffer using the UART0
    txDataBytes = XUartNs550_Send(&uart0, txHelloWorld,
    ↪  sizeof(txHelloWorld));

    xil_printf("The number of bytes sent from UART0 is %d\n\r",
    ↪  txDataBytes);

    sleep(1); // Add 1 second delay to confirm that data has
    ↪  been sent

    // Receive data from UART1 and store in buffer
    rxDataBytes = XUartNs550_Recv(&uart1, rxHelloWorld,
    ↪  sizeof(txHelloWorld));

    xil_printf("The number of bytes received by UART1 is %d\n\r",
    ↪  rxDataBytes);

    for(int i = 0; i < 5; i++){

            // Sending data which is received from UART0 to the
            ↪  UART-USB bridge connected to the computer
            XUartNs550_Send(&uart1, rxHelloWorld,
            ↪  sizeof(rxHelloWorld));
```

```
                // One second pause to make sure that entire data is
                ↪  sent.
                sleep(1);
        }

        return 0;
}
```

B.2 Interrupt Design

```
/*
 * This application makes sure to send/receive data between two
 * UART devices with the usage of interrupt signals. Also allowing
 * user to change the baud as desire.
 * PS7 UART (Zynq) is not initialized by this application, since
 * bootrom/bsp configures it to baud rate 115200
 *
 * ------------------------------------------------
 * | UART TYPE   BAUD RATE                        |
 * ------------------------------------------------
 *   uartns550   Whatever value the operator desires. Adjusted at
↪  line 35
 *   ps7_uart    This is the I/O UART1 peripheral on the Zynq PS
↪  ALWAYS set at 115200 baud rate
 */

/***************************** Include Files
↪  *********************************/

#include "xil_printf.h"
#include "xil_exception.h"
#include "xparameters.h"
#include "xscugic.h"
#include "xuartns550.h"

#include <stdio.h>
#include "sleep.h"

/************************** Constant Definitions
↪  *****************************/

#define UART_DEVICE_ID0                 XPAR_UARTNS550_0_DEVICE_ID
#define UART_DEVICE_ID1                 XPAR_UARTNS550_1_DEVICE_ID
#define
↪  UART_IRPT_INTR0                XPAR_FABRIC_AXI_UART16550_0_IP2INTC_IRPT_INTR
#define
↪  UART_IRPT_INTR1                XPAR_FABRIC_AXI_UART16550_1_IP2INTC_IRPT_INTR
#define INTC_DEVICE_ID                  XPAR_SCUGIC_SINGLE_DEVICE_ID

// Baud Rate, Highest possible value 2.32Mbps when having 250 MHz
↪  AXI CLK Frequency
#define UART_BAUDRATE                 2320000
#define DATA_BITS                         XUN_FORMAT_8_BITS
#define PARITY                                XUN_FORMAT_NO_PARITY
#define STOP_BITS                         XUN_FORMAT_1_STOP_BIT
```

```
// Buffer to be Tx from UART0, and buffer for Rx at UART1
//u8 SendBuffer[] = "Hi!";
u8 SendBuffer[] = "Hello World!";
u8 RecvBuffer[sizeof(SendBuffer)];

// The following constant controls the length of the buffers to be
↪  sent and received with the UART.
#define TEST_BUFFER_SIZE        sizeof(SendBuffer)

XUartNs550Format UARTFormat =
{
                UART_BAUDRATE,
                DATA_BITS,
                PARITY,
                STOP_BITS,
};

/************************** Type Definitions
↪  *****************************/

#define INTC                        XScuGic
#define INTC_HANDLER        XScuGic_InterruptHandler

/************************** Function Prototypes
↪  *****************************/

int UartInitializations(XUartNs550 *UartInstancePtr0, XUartNs550
↪  *UartInstancePtr1, u16 UartDeviceId0, u16 UartDeviceId1);

void Uart0SendHandler(void *CallBackRef, u32 Event, unsigned int
↪  EventData);

void Uart1RecvHandler(void *CallBackRef, u32 Event, unsigned int
↪  EventData);

int IntcInitFunction(u16 DeviceId, XUartNs550 *UartInstancePtrX,
↪  XUartNs550 *UartInstancePtrY);

int InterruptSystemSetup(XScuGic *XScuGicInstancePtr);

void UartNs550DisableIntrSystem(XScuGic *XScuGicInstancePtr, u16
↪  UartIntrId0, u16 UartIntrId1);

/************************** Variable Definitions
↪  *****************************/

XUartNs550 UartNs550Instance0;          /* Instance of the UART Device
↪  */
```

```
XUartNs550 UartNs550Instance1;        /* Instance of the UART Device
↪  */
INTC IntcInstance;                                  /* Instance of the
↪  Interrupt Controller */

/*
 * The following counters are used to determine when the entire
↪  buffer has
 * been sent and received.
 */
static volatile int TotalReceivedCount;
static volatile int TotalSentCount;
static volatile int TotalErrorCount;

/*****************************************************************************/
/**
*
* Main function to call the UartNs550 interrupt.
*
*
* @return        XST_SUCCESS if successful, otherwise XST_FAILURE.
*
*
******************************************************************************/
int main(void)
{
        // Size of data to be transmitted/received
        xil_printf("The size of the \"Hello World!\" being
        ↪  transmitted is %d bytes\n\r", TEST_BUFFER_SIZE);

        // Test data used to see if the UART-USB bridge is working
        ↪  correctly
        u8 Test[] = "This is a Test\n";

        // Counter used to inform the operator whenever data was
        ↪  not successfully transmitted/received
        u32 BadByteCount = 0;

        int Status;
        u16 Options;

        // Initialize the UARTs and self test each
        Status = UartInitializations(&UartNs550Instance0,
        ↪  &UartNs550Instance1, UART_DEVICE_ID0, UART_DEVICE_ID1);
        if (Status != XST_SUCCESS) {
                return XST_FAILURE;
        }
```

```
        Status = XUartNs550_SetDataFormat(&UartNs550Instance0,
        ↪  &UARTFormat);
        if (Status != XST_SUCCESS) {
                return XST_FAILURE;
        }

        Status = XUartNs550_SetDataFormat(&UartNs550Instance1,
        ↪  &UARTFormat);
        if (Status != XST_SUCCESS) {
                return XST_FAILURE;
        }

        /*
         * Printing the configuration set to both UART devices
↪   through the
         * PS7 UART on the ZedBoard
         */

        print("*****UART DATA FORMAT*****\n\r");

    printf("The baud is set to %.2f Mbps\n\r", (float)
    ↪  UARTFormat.BaudRate / 1000000);

    switch (UARTFormat.DataBits) {
                case 0:
                    print("The number of data bits is set to 5\n\r");
                        break;

                case 1:
                        print("The number of data bits is set to
                        ↪  6\n\r");
                        break;

                case 2:
                        print("The number of data bits is set to
                        ↪  7\n\r");
                        break;

                case 3:
                        print("The number of data bits is set to
                        ↪  8\n\r");
                        break;
                default:
                        print("The data bits were not appropriately
                        ↪  set.\n\r");
                        break;
        }
```

```
switch (UARTFormat.Parity) {
            case 0:
                    print("No parity has been chosen\n\r");
                    break;

            case 1:
                    print("Odd parity has been chosen\n\r");
                    break;

            case 2:
                    print("Even parity has been chosen\n\r");
                    break;

            default:
                    print("The parity was incorrectly set.\n\r");
                    break;
    }

if(UARTFormat.StopBits == 0){
        print("UARTs will have 1 stop bit\n\r");
} else {
        print("UARTs will have 2 stop bits\n\r");
}

print("***************************\n\r");

    // Send the test signal to the UART-USB bridge
    XUartNs550_Send(&UartNs550Instance1, Test, sizeof(Test));

    sleep(1);

    // Initialize interrupt controller for both UARTs
    Status = IntcInitFunction(INTC_DEVICE_ID,
     ↪  &UartNs550Instance0, &UartNs550Instance1);
    if(Status != XST_SUCCESS){
            return XST_FAILURE;
    }

    // Setup interrupt handlers for the UARTs that will be
     ↪  called once data is transmitted/received
    XUartNs550_SetHandler(&UartNs550Instance0, Uart0SendHandler,
     ↪  &UartNs550Instance0);
    XUartNs550_SetHandler(&UartNs550Instance1, Uart1RecvHandler,
     ↪  &UartNs550Instance1);

    // Enable data interrupt type, enable both Tx/Rx FIFOs and
     ↪  reset both FIFOs
```

```
Options = XUN_OPTION_DATA_INTR | XUN_OPTION_FIFOS_ENABLE |
↪  XUN_OPTION_RESET_TX_FIFO | XUN_OPTION_RESET_RX_FIFO;
Status = XUartNs550_SetOptions(&UartNs550Instance0, Options);
if (Status != XST_SUCCESS) {
        return XST_FAILURE;
}
Status = XUartNs550_SetOptions(&UartNs550Instance1, Options);
if (Status != XST_SUCCESS) {
        return XST_FAILURE;
}

// Choose UART1 to have a FIFO Threshold at 8 bytes
Status = XUartNs550_SetFifoThreshold(&UartNs550Instance1,
↪  XUN_FIFO_TRIGGER_08);
if (Status != XST_SUCCESS) {
        return XST_FAILURE;
}

// Make the UART1 ready to receive data (receive before
↪  send because of loopback) and save it into "RecvBuffer"
↪  buffer
XUartNs550_Recv(&UartNs550Instance1, RecvBuffer,
↪  TEST_BUFFER_SIZE);
//xil_printf("The amount of data received from UART0 is %d
↪  bytes\n\r", ReceivedCount1);

// Send the data from "SendBuffer" buffer using the UART0
XUartNs550_Send(&UartNs550Instance0, SendBuffer,
↪  TEST_BUFFER_SIZE);
//xil_printf("The amount of data sent from UART0 is %d
↪  bytes\n\r", SentCount0);

// Wait for the entire data to be received, while letting
↪  the interrupt processing work in the background
while ((TotalReceivedCount != TEST_BUFFER_SIZE) ||
↪  (TotalSentCount != TEST_BUFFER_SIZE)) {}

// Verify that data has been sent correctly without errors.
↪  If error has occurred "BadByteCount" will increment
for (int i = 0; i < TEST_BUFFER_SIZE; i++) {
  if (RecvBuffer[i] != SendBuffer[i]) {
    BadByteCount++;
  }
}

// Send data from UART1 to computer
XUartNs550_Send(&UartNs550Instance1, RecvBuffer,
↪  sizeof(RecvBuffer));
```

```
        // Disable the UART interrupts
        UartNs550DisableIntrSystem(&IntcInstance, UART_IRPT_INTR0,
        ↪  UART_IRPT_INTR1);

        // Notify the the operator about
        if(BadByteCount != 0){
                print("The correct data was not transmitted\n\r");
                return XST_FAILURE;
        }

        print("Data was successfully transmitted and received.\n\r");

        /* Clear the counters */
        TotalErrorCount = 0;
        TotalReceivedCount = 0;
        TotalSentCount = 0;

        /*
         * Clean up the options
         */
        Options = XUartNs550_GetOptions(&UartNs550Instance0);
        Options = Options & ~(XUN_OPTION_DATA_INTR |
        ↪  XUN_OPTION_FIFOS_ENABLE | XUN_OPTION_RESET_TX_FIFO |
        ↪  XUN_OPTION_RESET_RX_FIFO);
        XUartNs550_SetOptions(&UartNs550Instance0, Options);
        XUartNs550_SetOptions(&UartNs550Instance1, Options);

        return XST_SUCCESS;
}

int UartInitializations(XUartNs550 *UartInstancePtr0, XUartNs550
↪  *UartInstancePtr1, u16 UartDeviceId0, u16 UartDeviceId1) {
        int Status;

        // Initialize the UART drivers so that they're ready to
        ↪  use.
        Status = XUartNs550_Initialize(UartInstancePtr0,
        ↪  UartDeviceId0);
        if (Status != XST_SUCCESS) {
                return XST_FAILURE;
        }

        Status = XUartNs550_Initialize(UartInstancePtr1,
        ↪  UartDeviceId1);
        if (Status != XST_SUCCESS) {
                return XST_FAILURE;
        }
```

```
        // Perform a self-test to ensure that the UARTs was built
        ↪  correctly.
        Status = XUartNs550_SelfTest(UartInstancePtr0);
        if (Status != XST_SUCCESS) {
                return XST_FAILURE;
        }

        Status = XUartNs550_SelfTest(UartInstancePtr1);
        if (Status != XST_SUCCESS) {
                return XST_FAILURE;
        }

        return XST_SUCCESS;
}

void Uart0SendHandler(void *CallBackRef, u32 Event, unsigned int
↪  EventData)
{
        // All of the data has been sent
        if (Event == XUN_EVENT_SENT_DATA) {
                TotalSentCount = EventData;
                xil_printf("UART0 has sent %d bytes to UART1\n\r",
                ↪  TotalSentCount);
        }
}

void Uart1RecvHandler(void *CallBackRef, u32 Event, unsigned int
↪  EventData)
{
        u8 Errors;
        XUartNs550 *UartNs550Ptr = (XUartNs550 *)CallBackRef;

        // All of the data has been received
        if (Event == XUN_EVENT_RECV_DATA) {
                TotalReceivedCount = EventData;
                xil_printf("UART1 has received %d bytes from
                ↪  UART0\n\r",TotalReceivedCount);
        }

        /*
         * Data was received, but not the expected number of bytes
         */
        if (Event == XUN_EVENT_RECV_TIMEOUT) {
                TotalReceivedCount = EventData;
                print("Data which is received from UART0 is lower
                ↪  than threshold set at 8 byte\n\r");
        }
```

```
        // Data was received with an error, keep the data but
        ↪  determine what kind of error occured
        if (Event == XUN_EVENT_RECV_ERROR) {
                TotalReceivedCount = EventData;
                TotalErrorCount++;
                Errors = XUartNs550_GetLastErrors(UartNs550Ptr);

                switch (Errors) {
                case XUN_ERROR_BREAK_MASK:
                        print("ERROR: Break detected!");
                case XUN_ERROR_FRAMING_MASK:
                        print("ERROR: Frame error!");
                case XUN_ERROR_PARITY_MASK:
                        print("ERROR: Parity error!");
                case XUN_ERROR_OVERRUN_MASK:
                        print("ERROR: Overrun error!");
                }
        }
}

int InterruptSystemSetup(XScuGic *XScuGicInstancePtr)
{

        // Initialize the exception table.
        Xil_ExceptionInit();

        // Register the interrupt controller handler with the
        ↪  exception table.
        Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
        ↪  (Xil_ExceptionHandler)INTC_HANDLER, XScuGicInstancePtr);

        // Enable exceptions
        Xil_ExceptionEnable();


        return XST_SUCCESS;

}

int IntcInitFunction(u16 IntcDeviceId, XUartNs550 *UartInstancePtr0,
↪  XUartNs550 *UartInstancePtr1)
{
        XScuGic_Config *IntcConfig;
        int Status;

        // Interrupt controller initialisation making it ready to
        ↪  use
```

```
IntcConfig = XScuGic_LookupConfig(IntcDeviceId);
if (NULL == IntcConfig) {
        return XST_FAILURE;
}

Status = XScuGic_CfgInitialize(&IntcInstance, IntcConfig,
↪  IntcConfig->CpuBaseAddress);
if(Status != XST_SUCCESS){
        return XST_FAILURE;
}

// Setting priority and trigger to be rising edge sensitive
XScuGic_SetPriorityTriggerType(&IntcInstance,
↪  UART_IRPT_INTR0, 0xA0, 0x3);
XScuGic_SetPriorityTriggerType(&IntcInstance,
↪  UART_IRPT_INTR1, 0xB0, 0x3);

// Connect interrupt handler that will be called when an
↪  interrupt occurs for the UART0
Status = XScuGic_Connect(&IntcInstance, UART_IRPT_INTR0,
                (Xil_ExceptionHandler)XUartNs550_InterruptHandler,
                ↪  UartInstancePtr0);
if (Status != XST_SUCCESS) {
        return Status;
}

// Connect interrupt handler that will be called when an
↪  interrupt occurs for the UART1
Status = XScuGic_Connect(&IntcInstance, UART_IRPT_INTR1,
                (Xil_ExceptionHandler)XUartNs550_InterruptHandler,
                ↪  UartInstancePtr1);
if (Status != XST_SUCCESS) {
        return Status;
}

// Enable the interrupt for the UART0 device.
XScuGic_Enable(&IntcInstance, UART_IRPT_INTR0);

// Enable the interrupt for the UART1 device.
XScuGic_Enable(&IntcInstance, UART_IRPT_INTR1);

// Call to interrupt setup
Status = InterruptSystemSetup(&IntcInstance);
if(Status != XST_SUCCESS){
        return XST_FAILURE;
}

return XST_SUCCESS;
```

```
}

void UartNs550DisableIntrSystem(XScuGic *XScuGicInstancePtr, u16
↪  UartIntrId0, u16 UartIntrId1){

        // Disable and disconnect interrupt for the UARTs
        XScuGic_Disable(XScuGicInstancePtr, UartIntrId0);
        XScuGic_Disable(XScuGicInstancePtr, UartIntrId1);

        XScuGic_Disconnect(XScuGicInstancePtr, UartIntrId0);
        XScuGic_Disconnect(XScuGicInstancePtr, UartIntrId1);
}
```

Appendix C

Specialization Project VHDL Code

Source code for specialization project. **NOT** used in this project

C.1 Transmitter (tUART)

```
--------------------------------------------------------------------------------
--        FILE:                           tUART.vhd
--
--        DESCRIPTION:        This design is used to implement a
↪  UART transmitter.
--------------------------------------------------------------------------------

-- Libraries
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.numeric_std.ALL;

-- Entity Declaration
ENTITY tUART IS
GENERIC (
        baud      : INTEGER := 3e6;        -- 3Mbps  Desired baud
        ↪  rate                           CHANGEABLE
        clk_rate  : INTEGER := 100e6);     -- 100MHz (ZedBoard, clock
        ↪  speed)                 CHANGEABLE
PORT (
        data_out  : OUT STD_LOGIC;                    -- transmit
        ↪  output
        tx_ready  : OUT STD_LOGIC;                    -- indicating
        ↪  that transmit line ready for another byte of data
        start     : IN STD_LOGIC;                     -- transmit
        ↪  starter
        data_in   : IN STD_LOGIC_VECTOR(7 DOWNTO 0);  -- little
        ↪  endian, byte of data to be transmitted
```

```
        reset_n    : IN STD_LOGIC;                    -- active LOW
        ↪  reset signal
        clk        : IN STD_LOGIC);                   -- FPGA Clock
END tUART;

-- Architecture
ARCHITECTURE behavior OF tUART IS
--------------------------------------------------------------------------------
-- Constants
--------------------------------------------------------------------------------
CONSTANT clk_freq : INTEGER := clk_rate;
CONSTANT max_clk_count : INTEGER := clk_freq / baud;       -- number
↪  of clock cycles it takes to achieve 1-bit time, for a desired
↪  baud rate
CONSTANT max_transmission_delay : INTEGER := 60 / 10e6;    -- #
↪  clock cycles needed to achieve a desired delay between data
↪  bits
CONSTANT max_bits : INTEGER := 10;                         -- total
↪  number of bits in our UART data packet. 1 start, 8 data, 1
↪  stop.

--------------------------------------------------------------------------------
-- Signals
--------------------------------------------------------------------------------
-- Counter Signals
SIGNAL clk_counter : INTEGER RANGE 0 TO max_clk_count;
↪  -- counter signal counting clock cycles until reaching
↪  max_clk_count representing 1 bit of data
SIGNAL clk_delay_counter : INTEGER RANGE 0 TO max_transmission_delay;
↪  -- counter signal counting clock cycles until reachign
↪  max_transmission_delay representing desired delay between data
↪  bits.
SIGNAL number_bits : INTEGER RANGE 0 TO max_bits;
↪  -- count number of bits in UART transmit

-- Signals used for edge detection circuitry
SIGNAL start_count_lead : STD_LOGIC := '0';
SIGNAL start_count_follow : STD_LOGIC := '0';
SIGNAL start_trans : STD_LOGIC := '0';

-- Signals used for UART shift register
SIGNAL data_reg : STD_LOGIC_VECTOR(9 DOWNTO 0) := (OTHERS => '1'); --
↪  little endian data register to be used in shifting of data

-- SM control signals
SIGNAL transmit_done : STD_LOGIC := '0'; -- signal indicating
↪  transmission of data is done
```

```
SIGNAL load_data : STD_LOGIC := '0'; -- signal responsible for
↪  loading data from data_in port
SIGNAL delay_clock : STD_LOGIC := '0';
SIGNAL delay_clock_done : STD_LOGIC := '0';
SIGNAL done_shifting : STD_LOGIC := '0'; -- signal that will turn
↪  high when all of the data bits has been
↪  shifted
SIGNAL shift_data : STD_LOGIC := '0'; -- signal indicating that a
↪  bit is to be shifted
↪
SIGNAL tx_ready_reg : STD_LOGIC := '0';

-- State machine state definitions
TYPE state_type IS(init_state, load_state, shift);
SIGNAL state, nxt_state : state_type;

BEGIN
--------------------------------------------------------------------------------
--                          STATE MACHINE
--------------------------------------------------------------------------------
-- Two Step State Machine Process

-- Step One
-- Assigning signal "state" to the value of the signal "nxt_state"
state_proc : PROCESS (clk)
BEGIN
        IF rising_edge(clk) THEN
                IF (reset_n = '0') THEN
                        state <= init_state;
                ELSE
                        state <= nxt_state;
                END IF;
        END IF;
END PROCESS state_proc;

-- Indicate Tx is ready for data
tx_ready <= transmit_done AND tx_ready_reg AND NOT delay_clock;

-- Step Two
-- Evaluating multiple signals and determines what value to assign
↪  to the "nxt_state" signal
nxt_state_proc : PROCESS (state, start_trans, done_shifting,
↪  delay_clock_done)
BEGIN
        nxt_state <= state;
        load_data <= '0';
        transmit_done <= '0';
```

```
        CASE state IS
                WHEN init_state =>
                        transmit_done <= '1';
                        IF (start_trans = '1') THEN
                                nxt_state <= load_state;
                        ELSE
                                nxt_state <= init_state;
                        END IF;
                        IF (delay_clock_done = '1') THEN
                                delay_clock <= '0';
                        ELSE
                                delay_clock <= '1';
                        END IF;
                WHEN load_state =>
                        load_data <= '1';
                        nxt_state <= shift;
                WHEN shift =>
                        IF (done_shifting = '1') THEN
                                nxt_state <= init_state;
                                delay_clock <= '1';
                        ELSE
                                nxt_state <= shift;
                        END IF;
                WHEN OTHERS =>
                        nxt_state <= init_state;
        END CASE;
END PROCESS nxt_state_proc;
-------------------------------------------------------------------------------
--                          EDGE DETECTION
-------------------------------------------------------------------------------
start_trans <= start_count_lead AND (NOT start_count_follow);

begin_trans_proc : PROCESS (clk)
BEGIN
        IF (rising_edge(clk)) THEN
                IF (reset_n = '0') THEN
                        start_count_lead <= '0';
                        start_count_follow <= '0';
                ELSE
                        start_count_lead <= start;
                        start_count_follow <= start_count_lead;
                END IF;
        END IF;
END PROCESS begin_trans_proc;
-------------------------------------------------------------------------------
--
↪  COUNTERS
-------------------------------------------------------------------------------
```

```
-- Setting end of shift register to output
data_out <= data_reg(0);

-- Process that counts the number of bits shifted (transmitted)
-- Load data into the data_reg signal.
count_bits_proc : PROCESS (clk)
BEGIN
        IF (rising_edge(clk)) THEN
                IF (reset_n = '0') THEN
                        number_bits <= 0;

                ELSIF (number_bits = max_bits) THEN
                        done_shifting <= '1';
                        number_bits <= 0;

                ELSIF (load_data = '1') THEN
                        data_reg <= '1' & data_in & '0';
                        done_shifting <= '0';

                ELSIF (shift_data = '1') THEN
                        data_reg <= '1' & data_reg(9 DOWNTO 1);
                        number_bits <= number_bits + 1;
                END IF;
        END IF;
END PROCESS count_bits_proc;

-- Clock counters
clock_count_proc : PROCESS (clk)
BEGIN
        IF (rising_edge(clk)) THEN
                IF (state = shift) THEN
                        IF (clk_counter = max_clk_count) THEN
                                shift_data <= '1';
                                clk_counter <= 0;
                        ELSE
                                clk_counter <= clk_counter + 1;
                                shift_data <= '0';
                        END IF;
                END IF;
        END IF;
END PROCESS clock_count_proc;

-- Creates a delay after completion of transmission
clock_delay_proc : PROCESS (clk)
BEGIN
        IF (rising_edge(clk)) THEN
                IF (delay_clock = '1') THEN
```

```
                        IF (clk_delay_counter <
                        ↪  max_transmission_delay) THEN
                                clk_delay_counter <=
                                ↪  clk_delay_counter + 1;
                                delay_clock_done <= '0';
                        ELSE
                                delay_clock_done <= '1';
                        END IF;
                ELSE
                        clk_delay_counter <= 0;
                END IF;
        END IF;
END PROCESS clock_delay_proc;

-- Process placing transmit register busy when start signal is
↪  given
tx_ready_proc : PROCESS (clk)
BEGIN
        IF (rising_edge(clk)) THEN
                IF (start = '1') THEN
                        tx_ready_reg <= '0';
                ELSE
                        tx_ready_reg <= '1';
                END IF;
        END IF;
END PROCESS tx_ready_proc;

END behavior;
```

C.2 Testbench Transmitter (tb_tUART)

```
--------------------------------------------------------------------------------
--        FILE:                          tUART_sim.vhd
--
--        DESCRIPTION:        Simulates data transmitted at the
↪  following conditions:
--                                                  Baud
↪                   : 3 Mbps
--                                                  Data
↪                   : 8 bit
--
--        The following data sequence shall be transmitted:
--        'ALG'
--------------------------------------------------------------------------------

-- Libraries
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.numeric_std.ALL;

-- Entity
ENTITY tb_tUART IS
END;

-- Architecture
ARCHITECTURE test OF tb_tUART IS

-- UART Transmitter Design Declaration
COMPONENT tUART
        GENERIC (
                baud : INTEGER := 3e6;          -- 3Mbps  Desired
                ↪  baud rate
                clk_rate : INTEGER := 100e6);    -- 100MHz
                ↪  (ZedBoard, clock speed)
        PORT (
                data_out : OUT STD_LOGIC;
                tx_ready : OUT STD_LOGIC;
                start : IN STD_LOGIC;
                data_in : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
                reset_n : IN STD_LOGIC;
                clk : IN STD_LOGIC);
END COMPONENT;

-- Simulation Constants
CONSTANT baud_rate : INTEGER := 3e6;     -- 3Mbps  Desired baud rate
CONSTANT clock_rate : INTEGER := 100e6;  -- 100MHz (ZedBoard, clock
↪  speed)
```

```
-- Simulation signals
SIGNAL data_out_sim : STD_LOGIC := '1';
SIGNAL tx_ready_sim : STD_LOGIC := '0';
SIGNAL start_sim : STD_LOGIC := '0';
SIGNAL data_in_sim : STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL reset_n_sim : STD_LOGIC := '0';
SIGNAL clk_sim : STD_LOGIC := '0';

BEGIN

clk_sim <= NOT clk_sim AFTER 5 ns;

-- UART Transmitter Instantiation
DUT : tUART GENERIC MAP(baud_rate, clock_rate)
PORT MAP(
        data_out => data_out_sim,
        tx_ready => tx_ready_sim,
        start => start_sim,
        data_in => data_in_sim,
        reset_n => reset_n_sim,
        clk => clk_sim);

-- Simulate the stimulus's to our design
stimulus : PROCESS
BEGIN
        -- Pull reset high
        WAIT FOR 4 ns;
        reset_n_sim <= '1';

        WAIT FOR 20 ns;

        -- Transmit 'A'
        data_in_sim <= x"41";
        start_sim <= '1';
        WAIT FOR 60 ns;
        start_sim <= '0';

        WAIT FOR 4000 ns;

        -- Transmit 'L'
        data_in_sim <= x"4C";
        start_sim <= '1';
        WAIT FOR 60 ns;
        start_sim <= '0';

        WAIT FOR 4000 ns;
```

```
        -- Transmit 'G'
        data_in_sim <= x"47";
        start_sim <= '1';
        WAIT FOR 60 ns;
        start_sim <= '0';

        REPORT "UART Terminal Test Completed!";
        WAIT;
END PROCESS stimulus;
END test;
```

C.3 Receiver (rUART)

```
--------------------------------------------------------------------------------
--        FILE:                          rUART.vhd
--
--        DESCRIPTION:        This design is used to implement a
↪  UART receiver.
--------------------------------------------------------------------------------

-- Libraries
library ieee;
use ieee.std_logic_1164.all;        -- Gives possibility to use
↪  standard logic type
use ieee.numeric_std.all;           -- Gives us unsigned and signed

-- Entity Declaration
entity rUART is
generic (
    baud        : integer := 9600;                    -- desired
    ↪  baud rate          CHANGEABLE
    clk_rate    : integer := 100e6);                  -- Clock rate
    ↪  (very common)   CHANGEABLE
port (
    data_in     : in std_logic;                        -- input
    ↪  signal of received data (serially)
    reset_n     : in std_logic;                        -- reset
    ↪  signal, active LOW
    clk         : in std_logic;                        -- Clock
    data_out    : out std_logic_vector(7 downto 0);    -- Little
    ↪  endian representation of received data to the module
    data_valid  : out std_logic                        --
    ↪  indication that received data is valid
);
end rUART;

-- Architecture
architecture Behavioral of rUART is
--------------------------------------------------------------------------------
-- Constants
--------------------------------------------------------------------------------
constant clk_freq               : integer := clk_rate;
constant max_bit_count          : integer := clk_freq / baud;
↪  -- number of clock cycles it takes to receive a single bit
constant max_start_bit_count    : integer := max_bit_count / 2;
↪  -- counts the number of cycles to send the start bit ||  It's
↪  divided by 2 to account for timing variations, the actual data
↪  bits are what matters, i.e. the max_bit_count values.
```

```
constant max_bits                : integer := 10;
↪  -- amount of bits in our packet 10 = start + data + stop.
-------------------------------------------------------------------------------
-- Signals
-------------------------------------------------------------------------------
-- *Counter Signals*
signal start_bit_counter      : integer range 0 to
↪  max_start_bit_count - 1;       -- counter signal indicating when
↪  the proper amount of cycles has been passed for the start bit
↪  to be read
signal bit_counter            : integer range 0 to max_bit_count - 1;
↪  -- counter signal indicating when the proper amount of cycles
↪  has been passed for the other bits to be read
signal number_bits            : integer range 0 to max_bits - 1;
↪  -- counter signal keeping track of the amount of bits that has
↪  been read, THIS INCLUDES START+DATA+STOP BIT

-- *Edge detection*
signal start_proc             : std_logic;
↪  -- transition signal to READ_START_BIT_STATE, indicating that
↪  start bit has been received and is ready to be read
signal start_reg              : std_logic_vector(1 downto 0) :=
↪  (others => '0');

-- *UART shift register*
signal data_reg               : std_logic_vector(9 downto 0) :=
↪  (others => '0'); -- data register used for shifting in the data
↪  input

-- *State Indicator Signals*
signal read_start             : std_logic;      -- indicating we're
↪  at READ_START_BIT_STATE, reading start bit
signal reading_data           : std_logic;      -- indicating we're
↪  at READ_BITS_STATE, reading the data bits

-- *Transition Signals*
signal read_bit         : std_logic;    -- transition signal, from
↪  READ_START_BIT_STATE -> READ_BITS_STATE, indicating whenever a
↪  bit is read
signal done_reading     : std_logic;    -- tranistion signal, from
↪  READ_BITS_STATE -> DONE_READING_STATE, indication that the all
↪  the signals have been read

type state is (INIT_STATE, READ_START_BIT_STATE, READ_BITS_STATE,
↪  DONE_STATE);
signal curr_state, next_state : state;

begin
```

```
------------------------------------------------------------------------------
--                                  STATE MACHINE
------------------------------------------------------------------------------

-- SYNCHRONOUS PART
state_proc : process (reset_n, clk)
begin
    if (reset_n = '0') then
        curr_state <= INIT_STATE;
    elsif rising_edge(clk) then
        curr_state <= next_state;
    end if;
end process state_proc;

-- COMBINATORIAL PART
next_state_proc : process (curr_state, start_proc, read_bit,
↪  done_reading)
begin

    -- ASSIGNING DEFAULT VALUES
    next_state <= curr_state;                    read_start <=
    ↪  '0';
        reading_data <= '0';                        data_valid <=
        ↪  '0';

    case (curr_state) is
        when INIT_STATE =>
            if start_proc = '1' then
                next_state <= READ_START_BIT_STATE;
            else
                next_state <= INIT_STATE;
            end if;

        when READ_START_BIT_STATE =>
            read_start <= '1';
            if read_bit = '1' then
                next_state <= READ_BITS_STATE;
            else
                next_state <= READ_START_BIT_STATE;
            end if;

        when READ_BITS_STATE =>
            reading_data <= '1';
            if done_reading = '1' then
                next_state <= DONE_STATE;
            else
                next_state <= READ_BITS_STATE;
            end if;
```

```
        when DONE_STATE =>
            data_valid <= '1';
            if start_proc = '1' then
                next_state <= READ_START_BIT_STATE;
            else
                next_state <= INIT_STATE;
            end if;
        when others =>
            next_state <= INIT_STATE;
    end case;
end process next_state_proc;

-------------------------------------------------------------------------------
--                          EDGE DETECTION
-------------------------------------------------------------------------------
start_proc <= start_reg(1) and start_reg(0);

start_reg_proc : process(clk)
begin
    if(rising_edge(clk)) then
        if(reset_n = '0') then
            start_reg(1 downto 0) <= (others => '0');
        else
            start_reg(0) <= not data_in;
            start_reg(1) <= not start_reg(0);
        end if;
    end if;
end process start_reg_proc;
-------------------------------------------------------------------------------
--                                  COUNTERS
-------------------------------------------------------------------------------
-- Start bit counter
start_bit_counter_proc : process (clk)
begin
    if (rising_edge(clk)) then
        if ((reset_n = '0') OR (start_bit_counter =
        ↪  max_start_bit_count-1)) then
            start_bit_counter <= 0;
        elsif (read_start = '1') then
            start_bit_counter <= start_bit_counter + 1;
        else
            start_bit_counter <= 0;
        end if;
    end if;
end process start_bit_counter_proc;
```

```
-- Counter process counting the amount of clock cycles for each bit
↪  to be read
bit_counter_proc : process (clk)
begin
    if (rising_edge(clk)) then
        if ((reset_n = '0') OR (bit_counter = max_bit_count-1)) then
            bit_counter <= 0;
        elsif (reading_data = '1') then
            bit_counter <= bit_counter + 1;
        else
            bit_counter <= 0;
        end if;
    end if;
end process bit_counter_proc;

-- Number of bits read tracker
number_bits_proc : process (clk)
begin
    if (rising_edge(clk)) then
        if ((reset_n = '0') or (number_bits = max_bits-1) or
        ↪  (reading_data = '0')) then
            number_bits <= 0;
        elsif (bit_counter = max_bit_count-1) then
            number_bits <= number_bits + 1;
        end if;
    end if;
end process number_bits_proc;
--------------------------------------------------------------------------------
--                          INDICATORS
--------------------------------------------------------------------------------
-- Throwing a flag whenever a bit is read (both START and DATA)
read_start_bit_proc : process (start_bit_counter, bit_counter)
begin
    if ((start_bit_counter = max_start_bit_count-1) or (bit_counter =
    ↪  max_bit_count-1)) then
        read_bit <= '1';
    else
        read_bit <= '0';
    end if;
end process read_start_bit_proc;

-- Process indicating when all the bits has been read
done_reading_proc : process (number_bits)
begin
    if (number_bits = max_bits-1) then
        done_reading <= '1';
    else
        done_reading <= '0';
```

```
    end if;
end process done_reading_proc;
-------------------------------------------------------------------------------
--                          SHIFT REGISTER
-------------------------------------------------------------------------------
data_out <= data_reg(8 downto 1); -- Send the data section to the
↪  output

shift_reg_proc : process (clk)
begin
    if (rising_edge(clk)) then
        if (reset_n = '0') then
            data_reg <= (others => '0');
        elsif (read_bit = '1') then
            data_reg <= data_in & data_reg(9 downto 1); -- Right
            ↪  shift action MSB becoming whatever data_in is
        end if;
    end if;
end process shift_reg_proc;

end Behavioral;
```

C.4 Testbench Receiver (tb_rUART)

```
--------------------------------------------------------------------------------
--        FILE:                          rUART_sim.vhd
--
--        DESCRIPTION:        Simulates data received at the
↪  following conditions:
--
--                        Baud                    : 3 Mbps
--                                                  Data
↪                   : 8 bit
--
--                 The following data sequence shall be received:
--                 'ALG'
--------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.ALL;
use ieee.numeric_std.all;

use std.textio.all ;
use ieee.std_logic_textio.all ;

entity rUART_sim is
end rUART_sim;

architecture test of rUART_sim is

component rUART
generic(
baud                          : integer := 3e6;
clk_rate                : integer := 100e6);
port(
data_out                : out std_logic_vector(7 downto 0);
data_valid                : out std_logic;
data_in                        : in std_logic;
reset_n                        : in std_logic;
clk                              : in std_logic);
end component;

constant baud_sim              : integer := 3e6;
constant clk_rate_sim    : integer := 100e6;

signal data_in_sim             : std_logic := '1';
signal clk_sim                    : std_logic := '0';
signal data_valid_sim   : std_logic := '0';
signal data_out_sim     : std_logic_vector(7 downto 0) := x"00";
signal reset_n_sim             : std_logic := '1';
```

```
begin

clk_sim <= not clk_sim after 5 ns;

DUT:  rUART
    generic map(baud => baud_sim, clk_rate => clk_rate_sim)
        port map(data_out => data_out_sim,
                 data_valid => data_valid_sim,
                 data_in => data_in_sim,
                 reset_n => reset_n_sim,
                 clk => clk_sim);

Rx_proc         : process
        begin
        wait for 600 ns;

    -- Simulate "A" --> 0100 0001
    data_in_sim <= '0';     -- START
    wait for 333.3 ns;
    data_in_sim <= '1';
        wait for 333.3 ns;
    data_in_sim <= '0';
        wait for 333.3 ns;
    data_in_sim <= '0';
        wait for 333.3 ns;
    data_in_sim <= '0';
        wait for 333.3 ns;
    data_in_sim <= '0';
        wait for 333.3 ns;
    data_in_sim <= '0';
        wait for 333.3 ns;
    data_in_sim <= '1';
        wait for 333.3 ns;
    data_in_sim <= '0';
    wait for 333.3 ns;
    data_in_sim <= '1';  -- STOP

    wait for 10 us;

    -- Simulate "L" --> 0100 1100
    data_in_sim <= '0';     -- START
    wait for 333.3 ns;
    data_in_sim <= '0';
        wait for 333.3 ns;
    data_in_sim <= '0';
        wait for 333.3 ns;
    data_in_sim <= '1';
```

```
        wait for 333.3 ns;
    data_in_sim <= '1';
        wait for 333.3 ns;
    data_in_sim <= '0';
        wait for 333.3 ns;
    data_in_sim <= '0';
        wait for 333.3 ns;
    data_in_sim <= '1';
        wait for 333.3 ns;
    data_in_sim <= '0';
    wait for 333.3 ns;
    data_in_sim <= '1';  -- STOP

    wait for 10 us;

    -- Simulate "G" --> 0100 0111
    data_in_sim <= '0';     -- START
    wait for 333.3 ns;
    data_in_sim <= '1';
        wait for 333.3 ns;
    data_in_sim <= '1';
        wait for 333.3 ns;
    data_in_sim <= '1';
        wait for 333.3 ns;
    data_in_sim <= '0';
        wait for 333.3 ns;
    data_in_sim <= '0';
        wait for 333.3 ns;
    data_in_sim <= '0';
        wait for 333.3 ns;
    data_in_sim <= '1';
        wait for 333.3 ns;
    data_in_sim <= '0';
    wait for 333.3 ns;
    data_in_sim <= '1';  -- STOP
    wait;

end process Rx_proc;
end test;
```