Gjermund Sollien Øfsti

NIRCA MkII DevKit Firmware Development

Master's thesis in Electronics Systems Design and Innovation
Supervisor: Snorre Aunet
Co-supervisor: Amir Hasanbegovic

January 2022

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

NTNU
Norwegian University of Science and Technology

ideas
Integrated Detector Electronics AS

Gjermund Sollien Øfsti

NIRCA MkII DevKit Firmware Development

Master's thesis in Electronics Systems Design and Innovation
Supervisor: Snorre Aunet
Co-supervisor: Amir Hasanbegovic
January 2022

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

Norwegian University of
Science and Technology

Department of Electronic Systems

Master's thesis

NIRCA MkII DevKit Firmware Development

Author:
Gjermund Sollien Øfsti

January 2024

Abstract

The NIRCA MkII Development Board(NM2DB) is a PCB featuring NIRCA MkII(NM2), a controller and readout ASIC developed by IDEAS. The PCB also utilizes a Trenz FPGA System-on-Module(SoM) to control the ASIC, control on-board LDOs and for communication between the PCB and a Matrox Frame Grabber connected to a Camera Link interface. In this thesis a firmware for the Trenz SoM is developed using VHDL. The firmware has UART communication between the frame grabber and the PCB as well as SPI communication to and from the NM2. Results in this thesis show that the firmware outperforms the previous firmware targeting the same PCB. This was done by switching away from Ethernet TCP communication and use more optimized software in python for reading and writing data to and from the NM2DB. The final implementation has a UART interface running at 460 800 baud and an SPI interface running at 10 MHz making a full write and read of $\approx$ 4600 registers on NM2 in 2 seconds while older firmware and software took up to 45 seconds.

Sammendrag

NIRCA MkII Development Board(NM2DB) er en PCB med NIRCA MkII(NM2), en ASIC brukt til kontroll og utlesning av infrarøde sensorer. PCBen har også en Trenz FPGA "system-på-modul" brukt til kontroll av ASICen, LDOer og for kommunikasjon mellom PCBen og en Frame Grabber fra Matrox koblet til med en Camera Link kabel. Masteroppgaven involverer utvilking av firmware for Trenz modulen med VHDL og software for å kommunisere med NM2DB fra PC. Firmwaren har UART kommunikasjon mellom frame grabber og PCBen i tillegg til SPI kommunikasjon mellom til og fra NM2. Resultatene i oppgaven viser at firmwaren utklasser den tidligere implementasjonen brukt på samme PCB. Dette er gjort ved å bytte bort fra Ethernet TCP kommunikasjon til å bruke mer optimisert python-kode til å skrive og lese data til NM2DB. Den endelige implementasjonen bruker en UART port som kjører på 460 800 baud og en SPI med frekvens 10 MHz. En full lese og skriveoperasjon til ca. 4600 registre på NM2 tok 2 sekunder mens gammel TCP software og hardware brukte omtrent 45 sekunder.

Title: NIRCA MkII Devkit Firmware Development

Student: Gjermund Sollien Øfsti

Problem description: IDEAS has developed a controller and readout ASIC, the NIRCA MkII, which is aimed at readout of infrared sensors. The NIRCA MkII has 17-channels with 16-bit 12Msps ADCs, 8 analog outputs and a single-cycle FSM with a limited instruction set for digital waveform generation. This project covers development firmware and software for the NIRCA MkII development board (NM2DB). The work shall be comprised of coding of new functionality, as well as possible modifications to existing IP. The work shall be tested using the NM2DB.

Assignment proposer: Amir Hasanbegovic, Project manager / Senior IC Design Engineer

NTNU supervisor: Snorre Aunet, Professor, Department of Electronic Systems

Preface

Due to a pause in my studies the fall of 2020, my master's degree in Electronics Systems Design and Innovation at NTNU is delayed by one semester. I contacted IDEAS in December 2022 as they offered a project assignment for the fall semester of 2022 to see if the project was still possible. Gunnar Mæhlum(CEO of IDEAS) agreed to let me get a chance at designing a PCB with under the guidance of Amir Hasanbegovic. After the semester ended I spent the summer as an intern for IDEAS continuing the work done in the spring semester. During the summer, I finished the design of NM2DB and got 5 copies produced.

After moving to Oslo during the summer, I got the opportunity to continue with my master's thesis at IDEAS developing firmware for the FPGA on NM2DB using VHDL. This task was challenging me with a whole new aspect of electronic design, but has been very rewarding. I feel lucky to be working alongside experienced engineers for the whole semester while completing my master's degree.

I would like to thank Gunnar Mæhlum for setting aside resources and people to help me with my project. A special thanks to Amir for spending time helping me finishing my work. I would also like to thank Snorre Aunet for help writing the report and the people at the IDEAS office for being helpful in small and big tasks.

Gjermund Sollien Øfsti

Table of Contents

List of Figures vii

List of Tables x

1 Introduction 1

1.1 Requirements . . . 2

2 Relevant background 3

2.1 NIRCA MkII Development Board(NM2DB) . . . 3

2.2 NIRCA MkII ASIC(NM2) . . . 6

2.2.1 SPI interface . . . 6

2.2.2 Registers and memory . . . 7

2.2.3 Clocks and reset . . . 7

2.2.4 Serial TX interface . . . 8

2.3 Trenz module . . . 9

2.3.1 Field-programmable Gate Array(FGPA) . . . 9

2.4 Matrox frame grabber . . . 10

2.5 Camera Link . . . 10

2.5.1 Camera Link IP . . . 10

2.6 Existing Firmware . . . 10

2.6.1 NM2 RX IP . . . 10

2.7 Existing test software . . . 12

3 Design 13

3.1 System overview . . . 13

3.2 Control module . . . 14

3.3 Vivado Block design . . . 15

3.3.1 Configuration of Processor System . . . 15

3.3.2 Clocking Wizard . . . 15

3.4 Configuration of Inputs and Outputs . . . 17

3.4.1 Differential buffers . . . 17

3.4.2 Tri-state signals . . . 17

3.4.3 Port mapping 18
3.5 Serial Interface 20
3.5.1 NM2 SPI commands 21
3.5.2 LDO enable 21
3.5.3 NM2 Clock Select 22
3.5.4 Data Acquisition 22
3.6 Top module 23
3.6.1 Clocks and resets 23
3.7 UART 23
3.7.1 UART RX 24
3.7.2 UART handler 25
3.7.3 UART TX 27
3.8 SPI 28
3.8.1 SPI handler 28
3.8.2 SPI master 29
3.9 LDO controller 30
3.10 Modifications of NM2DB 32
3.10.1 Altered schematic 33

4 Results **34**
4.1 Test Overview 34
4.2 VHDL module simulations 34
4.3 Vivado Timing and Resource Utilization 36
4.4 Test setup 37
4.5 PySerial 38
4.6 UART verification 39
4.6.1 Verifying the UART protocol 39
4.6.2 LDO configuration write and read 40
4.6.3 AVDDH programming 40
4.6.4 NM2 Clock Select 42
4.7 SPI verification 43
4.8 NIRCA MkII ASIC control 44

4.8.1 Write register test 45
4.8.2 Configuration registers write and read 46
4.8.3 Coefficient and RAM write and read 46
4.8.4 SPI reset and IO read 47
4.9 Performance testing 47
4.9.1 Performance of previous firmware and software 49

5 Discussion **50**

Bibliography **52**

Appendix **53**

A Vivado simulations 53
B nm2_ser_config.py 54
C Test reports 62
C.1 TestFullConfig9600.txt 62
C.2 TestFullConfig115200.txt 63
C.3 TestFullConfig460800.txt 64

List of Figures

1 Typical use case for the NM2DB. 1
2 Block diagram of NM2DB. 3
3 Photo of NM2DB. The interfaces important are marked with white text. 4
4 Block diagram of the NIRCA MkII ASIC. 6
5 Bitfield description for SPI register 0 and 1. 8
6 SPI reset bitfield description. 9
7 Photo of the Trenz module[Tre23]. 9
8 Camera Link pinout[Wik23]. 11
9 Architectural overview of the Camera Link IP. 12
10 Architectural overview of the NM2 RX IP. 12
11 High level block diagram of system. 13

12 FPGA firmware block diagram. The control module is marked with stapled lines. . . . 14
13 Block design generated in Vivado. . . . 16
14 Clocking wizard block diagram 17
15 Differential output buffer. . . . 17
16 Differential input buffer. . . . 18
17 Tri-state buffer instantiation. . . . 18
18 Serial format for NM2DB UART interface. . . . 20
19 LDO enable data frame. . . . 21
20 External supply(AVDDH) data frame. . . . 21
21 UART format for single and consecutive bytes. . . . 24
22 Block diagram for UART RX. . . . 24
23 FSM for UART RX 25
24 Block diagram for UART Handler. . . . 25
25 FSM for handling UART RX module. . . . 26
26 FSM for handling UART TX module. . . . 27
27 Block diagram for UART TX. . . . 27
28 FSM for UART TX 28
29 SPI protocol. . . . 29
30 Block diagram for SPI handler. . . . 29
31 Conversion of UART to SPI data. . . . 30
32 Block diagram for SPI master. . . . 30
33 SPI master FSM. . . . 31
34 LDO controller block diagram. . . . 31
35 Photos showing patches done to wire SPI pins to PL I/O. . . . 32
36 FPGA_DOUT 0-3 are replaced with SPI signals. This is the schematic of NM2DB after patch. . . . 33
37 FPGA pins connected to SPI signals after patch. The SPI signals are connected to B35 pins L4 and L23. . . . 33
38 SPI master simulation. . . . 35
39 Top module simulation, LDO enable. . . . 35
40 Post-implementation timing summary. . . . 36

41 Post-implementation utilization. NM2RX and Camera Link IPs are not implemented and their signals not constrained. 37

42 Picture of lab test setup. NM2DB is connected to the PC with Frame Grabber with Camera Link Cables and a JTAG cable. The blue ribbon cable is used to measure digital signals. A power supply is connected to the power input. 37

43 Picture of NM2DB with connectors. Left side: two Camera Link connectors and a power supply. Top: JTAG. Right side: Digital probes. 38

44 PySerial example configuration. 38

45 PySerial example write. 39

46 Measurement of SerTFG's positive line. Read response by sending LDO read command(0x09 LSB first). 40

47 Python snippet for testing all LDO enable configurations. 40

48 Python snippet for testing all AVDDH0 configurations. 41

49 Test configuration of AVDDH0. LDO is enabled at 0 s and incremented by 0.1 V every 0.2 s 41

50 6 MHz NM2 reference clock. Positive differential signal. 42

51 Write Reg1 using SPI. Data written is Command(**110**) + Data(**10010**). 43

52 Read Reg1 SPI test. Data written is Command(**111**)+ignored bytes **00000**. Data received is **0x12** 44

53 Setup procedure for serial interface. 45

54 Write ODAC0 register using memory map. 45

55 Python loop for testing ODAC0 and ODAC2. 45

56 Measurement of ODAC0 and ODAC2 when writing system registers controlling the ODACs. The blue line is ODAC0 and the pink is ODAC2. The pink measurement is offset by ≈1.5V. 46

57 Write and read test of all configuration registers with baud rate 9600. 46

58 Write and read test of all coefficient registers with baud rate 9600. . . 47

59 Write and read test of all even RAM registers with baud rate 9600. . 47

60 Full write and read of all registers with baud rate 115200. 48

61 Python code for finding highest baud rate accepted by frame grabber. 48

62 SPI master simulation result. 53

63 SPI handler simulation result. 53

64 UART TX simulation result. 53

65 UART RX simulation result. 53

List of Tables

1 Project requirements. 2
2 External connectors. 4
3 Internal interfaces. 5
4 NM2 SPI command interface. 7
5 NM2 memory spaces. 8
6 FPGA port constraints. 18
7 NM2 reference clock select. 22
8 Test overview for firmware. 34
9 Simulation results from Vivado. E = Enable bit, T = Toggle bit, D = Data bit, L = Length bit. 36
10 LDO control test procedure. 41
11 NM2 reference clock selection. 42
12 Default reg0 setup. 44
13 Default SPI reg1 setup. 44
14 Speed comparison of write and read from all memory spaces. 48

Glossary

ADC Analog to Digital Converter. 7

ASIC Application-specific integrated circuit. 1

BGA ball grid array. 9

IDEAS Integrated Detector Electronics AS. 1

IP Intellectual Property. In Vivado used as a functional block.. 15

MIL Matrox Imaging Library. 10

MIO Multiplexed Input/Output. 32

NM2 NIRCA MkII ASIC. 6

PL Programmable Logic. 15

PLL Phase-locked loop. 7

RAM Random-Access Memory. 7

RTL Register-Transfer Level. 15

SoC System on a Chip. 9

SPI Serial Peripheral Interface. 6

SWaP-C Size, Weight, Power and Cost. 1

VHDL VHSIC Hardware Description Language. 13

1 Introduction

NIRCA MkII is an Application-specific integrated circuit (ASIC) developed to reduce the Size, Weight, Power and Cost (SWaP-C) of video electronics in earth observation spacecraft payloads. Traditionally, video electronics have made use of many discrete components to cover all the required functions, which in turn is driving up the power consumption and making them bulky.

As part of the ASIC development, Integrated Detector Electronics AS (IDEAS) has developed a validation test system used for characterizing the ASIC. Although the system is good for ASIC characterization, it has several drawbacks. The system is unfit as a development kit due to having many test interfaces dedicated towards ASIC validation. The system also suffers from poor data throughput and slow configuration via the Ethernet interface due to firmware and software not being optimized for supporting an ASIC such as the NIRCA MkII.

To create a more user friendly and compact system, the NIRCA MkII Development Board(NM2DB) was created in the spring of 2023 as a part of the master's project[Øfs23]. NM2DB removed a lot of the test interfaces used to validate the ASIC and simplified the I/O by adding a compact sensor interface. Still there was a need for a faster and more user friendly firmware to give customers a rapid integration of the ASIC. This is where the firmware for this thesis comes into play.

The system developed in this thesis has a goal of removing the need for an Ethernet interface while reaching higher throughput, both for the configuration of NM2 as well as the data from NM2 to the Camera Link connector. A typical use case for NM2DB is shown in Figure 1. One or more sensors are connected to the sensor interface of NM2DB reading up to 16 channels of analog data while providing supply/reference voltages and a digital interface. NM2DB features a NM2 ASIC as well as an FPGA SoM used to implement the firmware interfacing with the ASIC. On the user end of the system, a PC with a Frame Grabber is connected with two Camera Link cables.

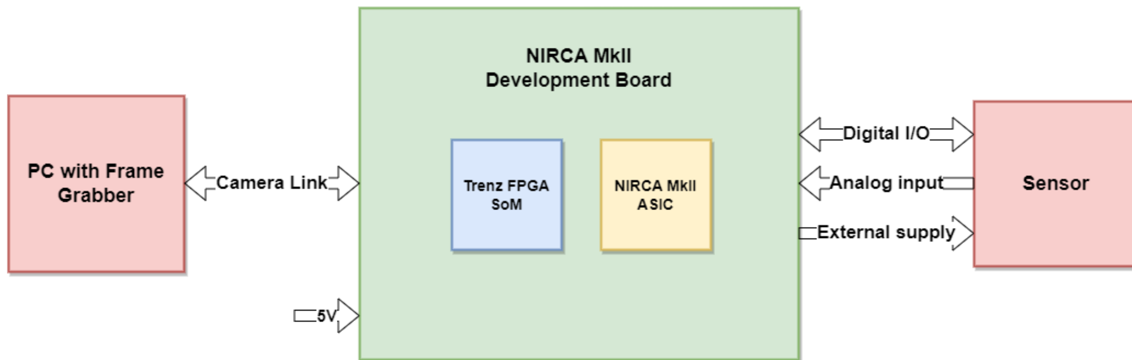


Figure 1: Typical use case for the NM2DB.

The new firmware implemented on the FPGA SoM using VHDL reduces the write of all NM2 registers from about 10 minutes to just under 2 seconds. This is done by using two serial ports in the Camera Link interface as a full duplex UART interface operating at a rate of 406 800 baud/s.

1.1 Requirements

A list of requirements is given as part of the master's project description and is presented in the list below. These requirements are the system minimum and should be met to fulfill the task.

Requirement ID	Requirement
REQ 1	The system shall be programmed via serial communication form the Camera Link interface. Both write and read operations shall be supported.
REQ 2	Camera Link shall be used for both configuration and image data transmission from FPGA to the frame grabber in the PC.
REQ 3	The system shall have an SPI which is compatible with the NIRCA MkII to be able to program the ASIC.
REQ 4	The system shall be able to program/control proximity modules on NM2DB, such as LDOs and NIRCA MkII IRQs.
REQ 5	NIRCA MkII shall interface to the NM2RX IP. Student shall instantiate the IP, make the necessary connections and modifications to the IP to allow data capture from the NIRCA MkII ASIC.
REQ 6	The NM2RX shall interface to the CAMERA LINK IP for data transport to the frame grabber in the PC. The student shall instantiate the IP, make the necessary connections and modifications to the IP to meet the requirements in this task.
REQ 7	Software shall be written (preferably in Python) that allows configuring the NIRCA MkII ASIC, the LDOs on NM2DB, and starting and image acquisition and capturing and storing data to a file.
REQ 8	The system shall be developed with future expansion in mind. Emphasis shall be on developing a well-structured system.
REQ 9	Target FPGA is Zynq-7020.
REQ 10	Target frame grabber is Matrox Radient eV-CL.
REQ 11	The Camera Link IP IO usage on the FPGA shall be modified to comply with the NM2DB.
REQ 12	The work shall be performed using well-structured and readable HDL code with sufficient comments.
REQ 13	The work shall be tracked at IDEAS Github.
REQ 14	The design shall be implemented and verified by using the NM2DB which has a Trenz TE0720 SOM with a Zynq-7020 and Camera Link hardware interface. The verification can be performed at IDEAS using the Matrox Radient eV-CL frame grabber.
REQ 15	(Bonus) Write a Python script for the frame grabber that performs configuration of LDOs and NM2. The script shall initiate a video readout (resolution = 1280x720, 200 frames per second, 10+ frames) and store the data to a binary file.

Table 1: Project requirements.

2 Relevant background

The relevant background section is intended to give the reader an overview in what work is already done related to this project, by the author, people at IDEAS or by external people.

2.1 NIRCA MkII Development Board(NM2DB)

During the spring of 2023 as a part of the master's project, the author of this thesis, Gjermund Øfsti, developed a development board intended for optimizing the previously developed test card (by IDEAS) towards a development kit for rapid adoption of the ASIC. This is the hardware that will be used for the final testing and demonstration for this thesis.

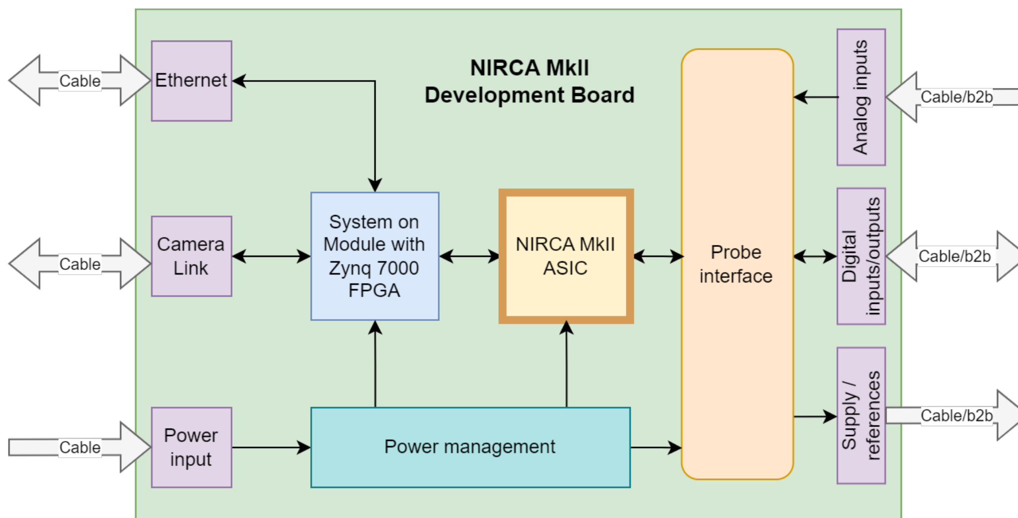


Figure 2: Block diagram of NM2DB.

The development board shown in Figure 2 and Figure 3 is based around the NIRCA MkII ASIC described in Section 2.2 and a Trenz Zynq 7000 system on module(SoM).

The power management consists of 6 LDOs used to power the on-board components as well as 2 configurable LDOs called AVDDH0 and AVDDH1 used for external supply via the sensor interface. The LDOs supplying the Trenz SoM are powered when the input voltage is connected, while the LDOs supplying the NIRCA MkII ASIC are enabled by the I/O of the Trenz SoM.

There are several interfaces, both external and internal, that are relevant for this thesis. These interfaces are briefly described in Table 3 and Table 2.

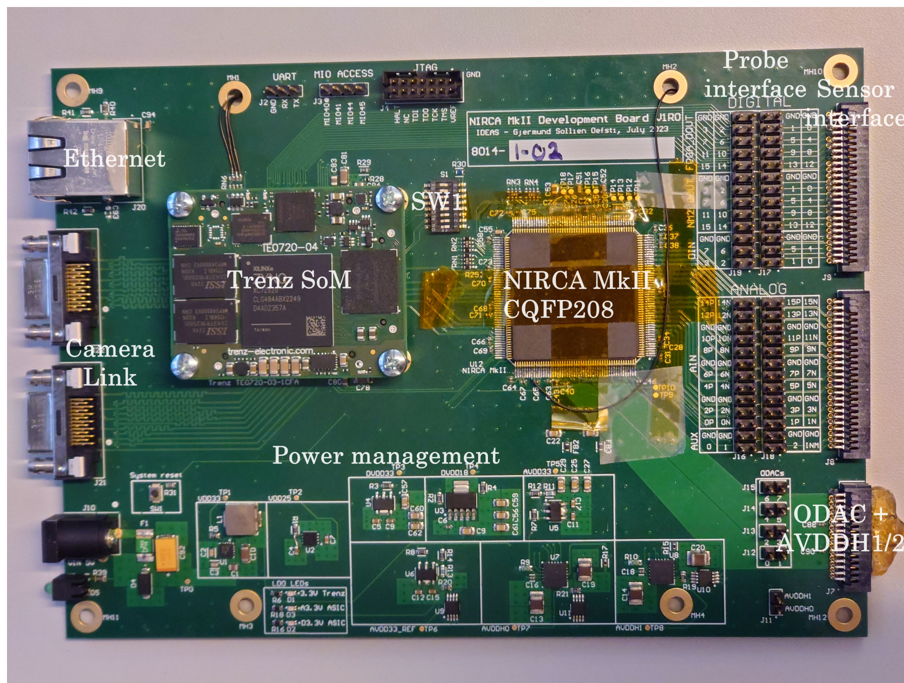


Figure 3: Photo of NM2DB. The interfaces important are marked with white text.

Connector	Description
Ethernet 8-pin RJ45 connector	To/from Trenz SoM The ethernet connector shall be used to retrieve data as well as communicate with the Trenz module.
2 Camera Link connectors	From Trenz SoM 26-pin Camera Link port with 4 ground pins and 22 signal wires connected directly to the Trenz Module. The Camera Link connectors shall be used to retrieve high speed data from the Trenz Module.
120-pins Pitch: 0.05”(0.127mm)	To/from NM2 and Trenz SoM The sensor interface of the development card shall have at least 120 pins including 96 signal and power pins.
8-pin tactile switch	From 2.5V to Trenz SoM Used to drive 8 input pins high for setting input to the FPGA SoC.
JTAG 14-pin connector	To/from Trenz SoM 7 pins connected to ground and 5 pins connected to Trenz for programming the SoM.
UART 3-pin connector	To/from Trenz SoM Provides a UART debug interface to the Trenz.
Probing interface Pitch: 0.1”(0.254mm)	To NM2 and Trenz SoM This probing interface is intended to be able to probe all the inputs in the sensor interface while connected. The 0.1” pitch is to have easier access to each pin.

Table 2: External connectors.

Type	Description
TX 20-pin	From NM2 to Trenz SoM 9 differential pairs for data transmission from the ASIC to the Trenz. The last pair provides a differential clock for the TX-lines.
SPI 4-pin	Between Trenz SoM and NM2 SPI interface from the Trenz to NM2 for programming the ASIC.
SCAN 5-pin	Between Trenz SoM and NM2 Test interface for the ASIC.
CLK 2-pin	From Trenz SoM to NM2 2-pin differential clock reference provided by the Trenz.

Table 3: Internal interfaces.

2.2 NIRCA MkII ASIC(NM2)

NIRCA MkII ASIC (NM2) is a controller and readout ASIC in development by IDEAS. The current version used here is a preliminary version of the final product and is described in detail in the NIRCA MkII datasheet[1]. The ASIC has 17 16-bit ADC channels on the sensor side and 9 TX channels capable of up to 480 Mbit/s upstream data transmission. To program the ASIC, an SPI interface is provided with eight system functions.

A block diagram for NM2 is shown in the Figure 4. The NM2 is a complex ASIC with many features that are out of scope for this thesis. The relevant part of NM2 is explained in this section and focuses mainly on the register read and write functionality.

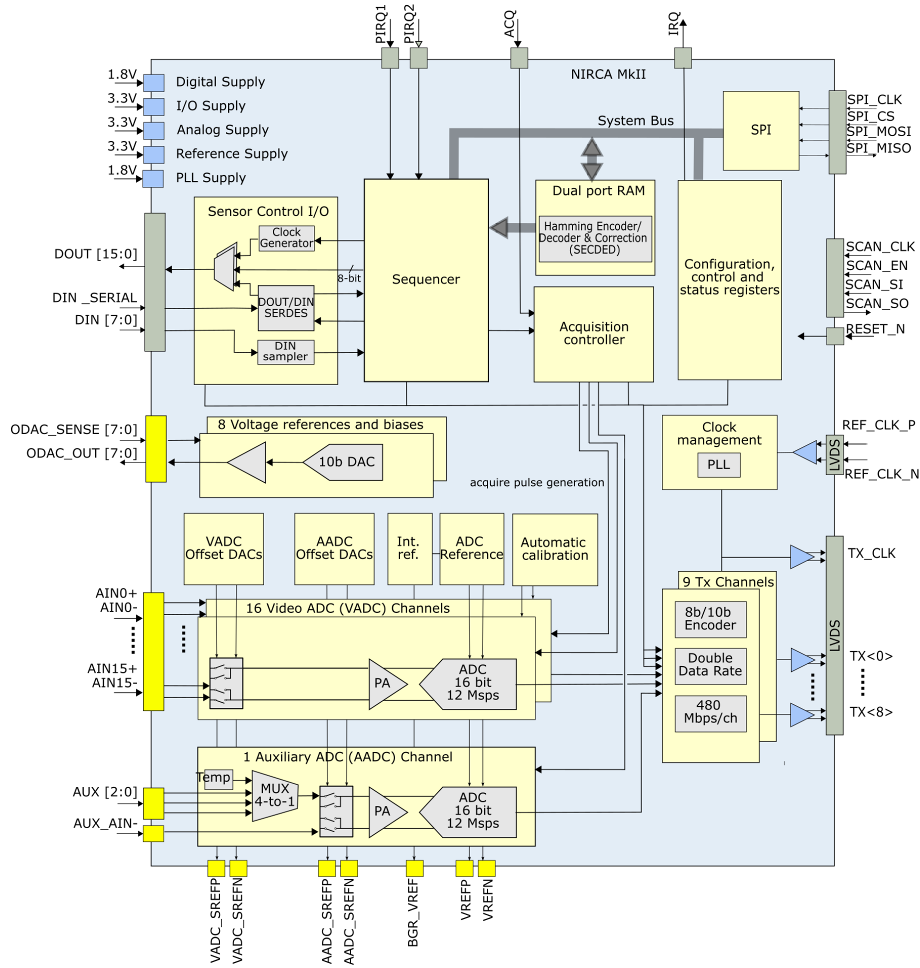


Figure 4: Block diagram of the NIRCA MkII ASIC.

2.2.1 SPI interface

The Serial Peripheral Interface (SPI) interface on the NM2 has 8 different functions described in Table 4. The SPI interface is operating in slave mode and with a clock polarity CPOL = 1 and clock phase CPHA = 1. This means that the clock signal

[1]Data sheet is an internal IDEAS document. Can be supplied on demand.

is idle when high and that data is sampled on the rising edge and shifted on the falling edge of the SPI clock.

SPI function	Command byte	Description
System read	**000PLLLL**	Send command byte followed by a 16bit address. The P is a prio flag to give priority on the system bus. LLLL is a 4bit length parameter indicating how many bytes to read from the provided address.
System write	**001PXXXX**	Send command byte followed by a 16bit address and a 8bit data byte. The data byte is written to the address provided.
IO read	**010IIIII**	Send command byte. The IO read has a 5bit selector to monitor pins on the MISO(Master In Slave Out).
SPI reset	**011FSSSS**	Send command byte. The fifo reset(F) generates a fifo reset if the system reset(S) is 0110 followed by 0000.
Reg0 write	**100DDDDD**	Write 5bit to SPI Reg0.
Reg0 read	**101XXXXX**	Read SPI Reg0.
Reg1 write	**110DDDDD**	Write 5bit to SPI Reg1.
Reg1 read	**111XXXXX**	Read SPI Reg1.

Table 4: NM2 SPI command interface.

2.2.2 Registers and memory

There are four types of memory spaces on the NM2 ASIC. These are SPI registers, system registers, instruction memory and ADC calibration registers.

The SPI registers consists of two 5bit registers used to control the bits shown in Figure 5. These registers are accessible directly through the SPI interface and via the system bus.

The other three memory spaces are accessed through the system bus. To enable the system bus, the *pll_enable* in SPI reg1 has to be set to **1**. When enabled, the registers are addressable with 16-bit addresses using **system read** and **system write**. The system memory space contains 236 bytes used for configuration, control and status of the NM2 ASIC. The instruction Random-Access Memory (RAM) memory is used for storing 15bit sequencer instructions and is 4096 bytes. Lastly the Analog to Digital Converter (ADC) calibrations memory has 32 addresses allocated for each of the 16 ADCs in NM2, but only the 19 first addresses for each ADC is used. An overview of the memory spaces is given in Table 5.

2.2.3 Clocks and reset

NM2 has two clock inputs. One is the SPI clock as part of the SPI interface which has a maximum frequency of 20 MHz. The other is the reference clock used for the internal Phase-locked loop (PLL) to generate the system clock for NM2. This clock has a maximum frequency of 15 MHz. The system reset is an active low signal used to reset all internal modules expect for the SPI interface.

The SPI interface has a reset function with bitfields explained in Figure 6.

Table 30: SPI_REG0 bitfield description.

Bit	Name	Reset	Comment
4:3	Reserved		
2	seq_halt	0	Active high halt command for the sequencer. Assert seq_halt before resetting and programming the sequencer to ensure default state of the sequencer.s
1	seq_reset	1	Active high synchronous reset for sequencer. Static register, needs to be written twice for reset/reset-release.
0	sys_clk_enable	0	Active high enable signal for clock gating module in digital domain.

Table 31: SPI_REG1 bitfield description.

Bit	Name	Reset	Comment
4	pll_enable	0	Active high PLL enable. '1'= PLL enabled. '0'= PLL disabled, reference clock bypassing PLL. PLL powered down.
2:3	sysclk _dly	00	sysclk_dly sets the sys_clk delay compared to clk4adc: sysclk_dly = "00" - sys_clk is 1 TX_clk_period delayed sysclk_dly = "01" - sys_clk is 2 TX_clk_period delayed sysclk_dly = "10" - sys_clk is 3 TX_clk_period delayed sysclk_dly = "11" - sys_clk is 4 TX_clk_period delayed
0:1	clk_div_mode	00	clk_div_mode: 00 => clk4adc : reference clock sys_clk : reference clock 01 => clk4adc : reference clock sys_clk : reference clock 10 => clk4adc : adc_clk_div (div 20) sys_clk : sys_clk_div (div 20) 11 => clk4adc : adc_clk_div (div 10) sys_clk : sys_clk_div (div 10)

Figure 5: Bitfield description for SPI register 0 and 1.

Memory space	Start address	End address	Description
System registers	0	235	236 addresses used for configuration, control and status bits. Each of the registers are either read/write, read only, read/reset or write/pulse.
Instruction memory	1024	1568	4096 addresses used for the instruction RAM. Instructions are 15bit, and stored as 7:0 in address 0 and 14:8 in address 1...
ADC calibration	8192	12288	ADC calibrations coefficient memory. Each ADC has 32 allocated addresses, but only use 19. ADC0 has addresses 1024-1024 +31, while ADC1 has addresses 1024+32-1024+63...

Table 5: NM2 memory spaces.

2.2.4 Serial TX interface

The serial TX interface is a interface transmitting 8b/10b encoded data over LVDS. The TX channels are used to either transmit the sampled data from the analog inputs. This interface will not be used in this firmware, but is connected to the NM2RX IP when implemented.

Bit	Name	Reset	Comment
4	fifo_reset	0	1 => Generates and holds SPI "Write FIFO" reset. 0 => Deactivates "Write FIFO" reset. system_reset must be "0110" for the above to have effect on the "Write FIFO".
0:3	system_reset	0	Asynchronous reset that goes to all modules. 0110 => Generates and holds system reset. 0000 => Deactivates system reset. To generate a reset pulse, two consecutive SPI_RESET commands are needed, i.e., DATA = 0110 and DATA = 0000.
SPI_RESET data is not stored in a addressable register and can therefore not be read.			

Figure 6: SPI reset bitfield description.

2.3 Trenz module

Both the Test Card and the NM2DB has a Trenz module[Tre23] with a Xilinx Zynq 7 series FPGA SoC. A photo of the front and back side of the module is shown in Figure 7. The Trenz module has 3 connectors on the back side with receiving connectors on NM2DB which have all the I/O pins from the FPGA as well as power supply pins and JTAG pins used to program the FPGA.

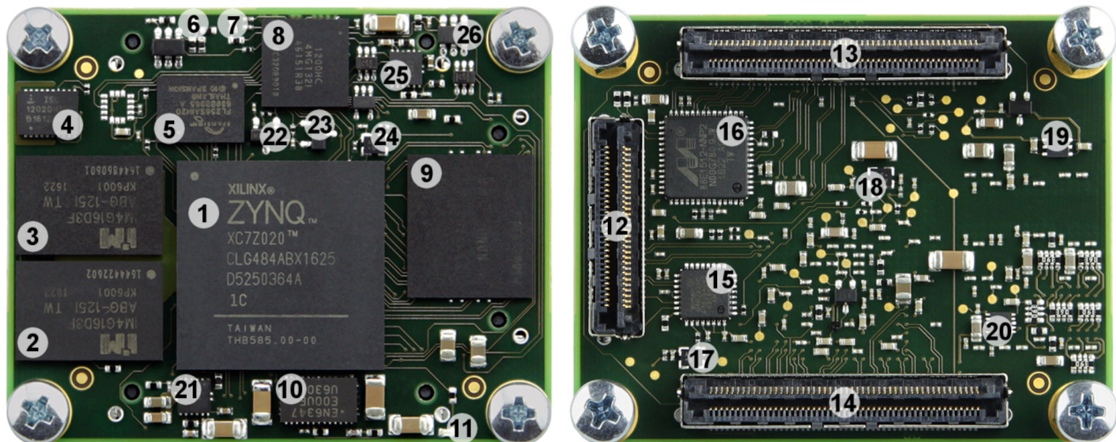


Figure 7: Photo of the Trenz module[Tre23].

2.3.1 Field-programmable Gate Array(FGPA)

The key part of this module is the Zynq 7 series FPGA System on a Chip (SoC). The firmware developed in this thesis is implemented in the FPGA. Underneath the FPGA SoC there is a Ball grid array (BGA) of 22x22(484) pins used for input/output(I/O) and power. The I/O pins are arranged in four banks with their own voltage supply. Bank 13 and 34 are supplied with 2.5V and are used for 2.5V LVDS signaling. The other banks, 33 and 35, are supplied with 3.3V and are used for other I/O.

2.4 Matrox frame grabber

A Matrox Radient eV-CL[Mat21] is connected to the NM2DB via the Camera Link connectors and to a PC motherboard via PCIe. The frame grabber is shipped with a software library called Matrox Imaging Library (MIL) which can be used to interface with the Camera Link connectors.

2.5 Camera Link

The Camera Link[AIA18] interface on the NM2DB is in full configuration as shown in Figure 8. Each data interface, X, Y and Z, has four LVDS signal pairs used for data transmission in addition to one signal pair used as a clock for each interface. There are four signal pairs used as camera control from the frame grabber to the camera. The last two signal pairs are serial lines reserved for UART communication.

2.5.1 Camera Link IP

Spring of 2023, Ole Tobias Moen wrote his master's thesis at IDEAS and developed a Camera Link encoder for an FPGA. The IP converts output from the NM2 to the Camera Link standard for use with a Matrox Frame Grabber. Figure 9[2] shows the architecture of the Camera Link IP.

The Camera Link interface on the NIRCA MkII development board is in full configuration meaning that it uses two 26pin connectors. There are 11 signal pairs(LVDS) on each connector. 15 of these pairs are used for data transmission, 4 pairs used for camera control and 2 pairs dedicated to a full-duplex UART interface.

2.6 Existing Firmware

IDEAS now uses a firmware called "DOPPIO" targeting the Trenz module and is customized to each PCB. This firmware utilizes an Ethernet port for communicating with the IDEAS Testbench software developed by IDEAS. The current implementation wraps each command in byte packets which are sent to the Trenz module's processor system and then passed via SPI to the NIRCA MkII ASIC. While this implementation works, it is reported to be relatively slow for applications where extensive SPI interaction is required. It also needs an Ethernet interface on top of the Camera Link used to extract data.

2.6.1 NM2 RX IP

The NM2 RX IP is developed by IDEAS and is intended to sample data from the upstream TX channels from the NM2 to the Trenz module. The architecture of the module is shown in Figure 10[3].

[2]Copied from Ole Tobias's master thesis. Can be supplied if necessary

[3]Figure is copied from an internal IDEAS document. Can be supplied if necessary.

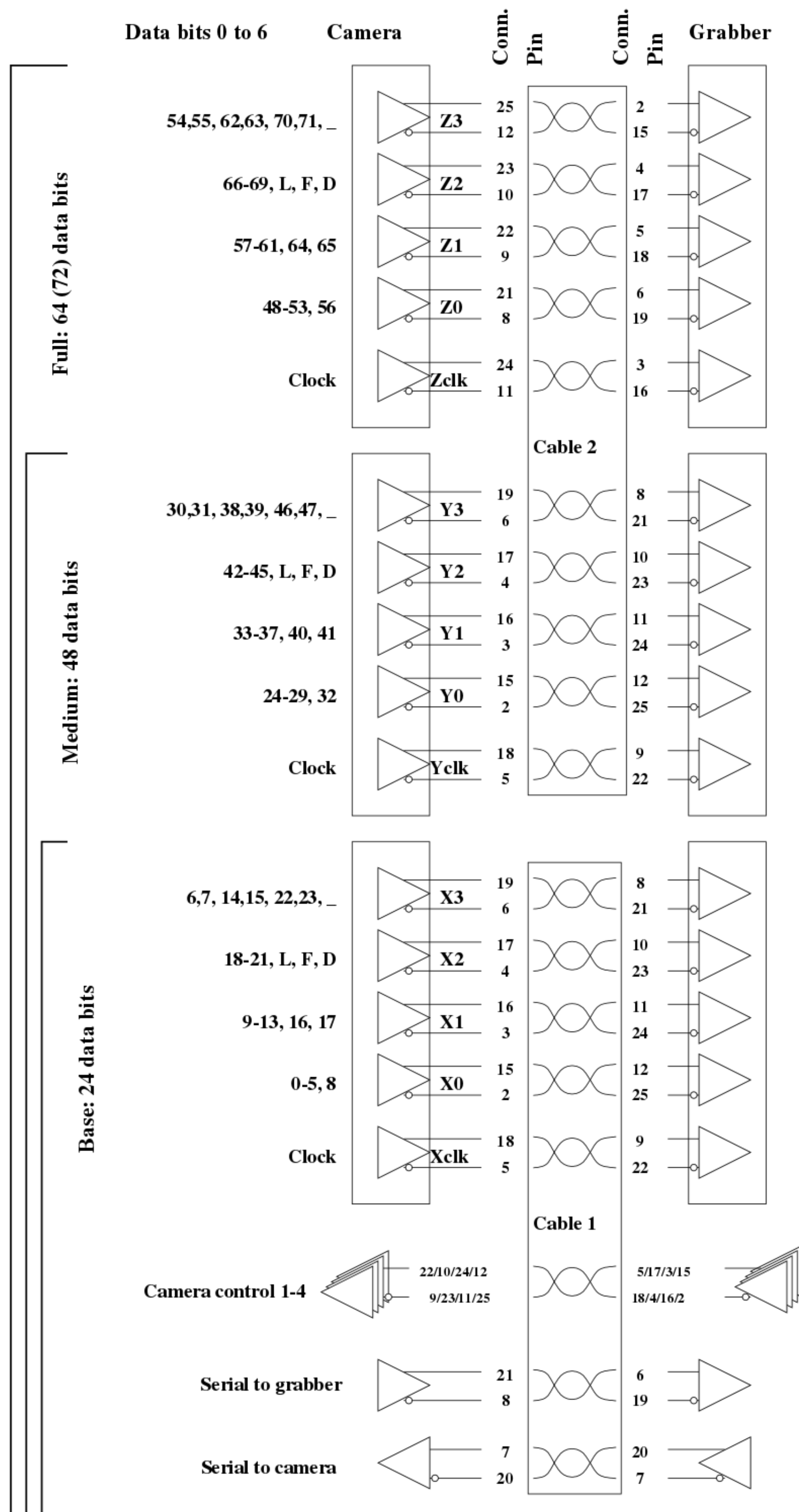


Figure 8: Camera Link pinout[Wik23].

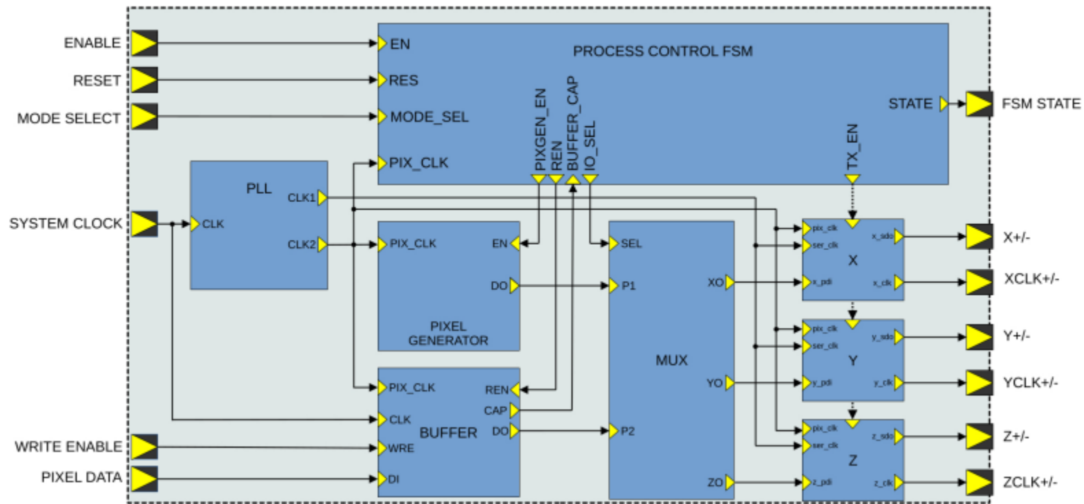


Figure 9: Architectural overview of the Camera Link IP.

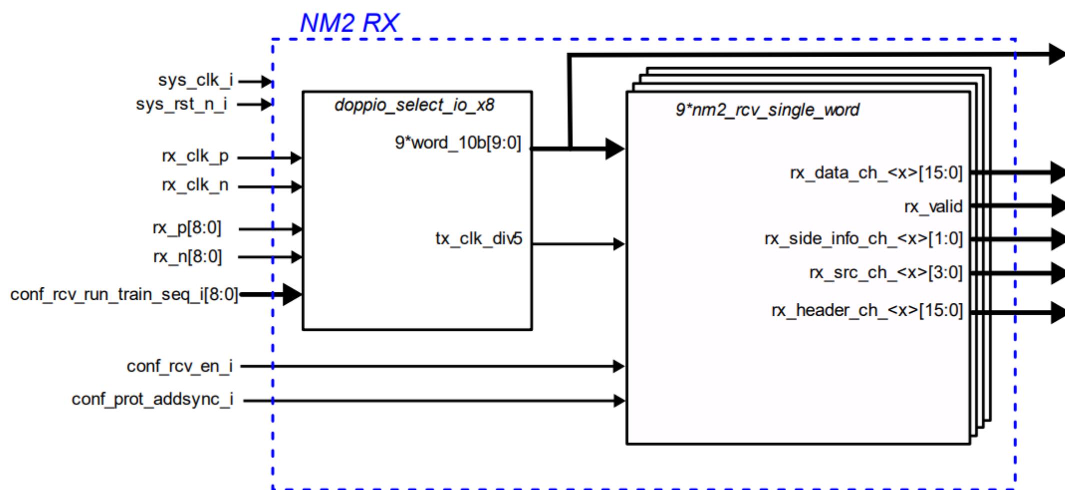


Figure 10: Architectural overview of the NM2 RX IP.

2.7 Existing test software

IDEAS has developed their own test interface called IDEAS Testbench. The software is a graphical interface used to read and write registers on the NIRCA MkII ASIC. Communication between the IDEAS Testbench and the Trenz SoM is done using Ethernet TCP. The Trenz SoM then communicates with NM2 using SPI.

In addition to the IDEAS Testbench, a Python library with functions targeting the NIRCA MkII ASIC is created. The functions are based on a memory map copying the memory on the ASIC. When resetting the ASIC, all configurations will be in a known state. Each write and read operation will update the memory map accordingly and the user will have a full overview of all registers.

3 Design

The firmware created in this thesis is a control system for the NM2DB meeting the requirements given in Section 1.1. The control system is implemented using VHSIC Hardware Description Language (VHDL) and compiled using Vivado 2023.1. The Vivado software offers a full design suite with several features used in this development. The design process consists of writing HDL modules covering the functionality needed and then writing test benches before testing the system in the real world. The test benches are used to verify the proper functionality of each module in the system. The Vivado software also offers an I/O-planning tool which is used to constrain port with voltage levels and port mapping to the external FPGA pins/pads.

The Design section is intended to give an overview of how the system is implemented with all the interfaces and functionality on the FPGA. The first sections 3.1-3.5 explains how the system is structured and configured to meet the specifications. Section 3.5 is describing how the serial interface can be seen from an user perspective. Sections 3.6-3.9 are intended to show how each module in the system operates on their own and in the system as a whole. Lastly section 3.10 is describing modifications done to NM2DB to adapt the PCB to this work.

3.1 System overview

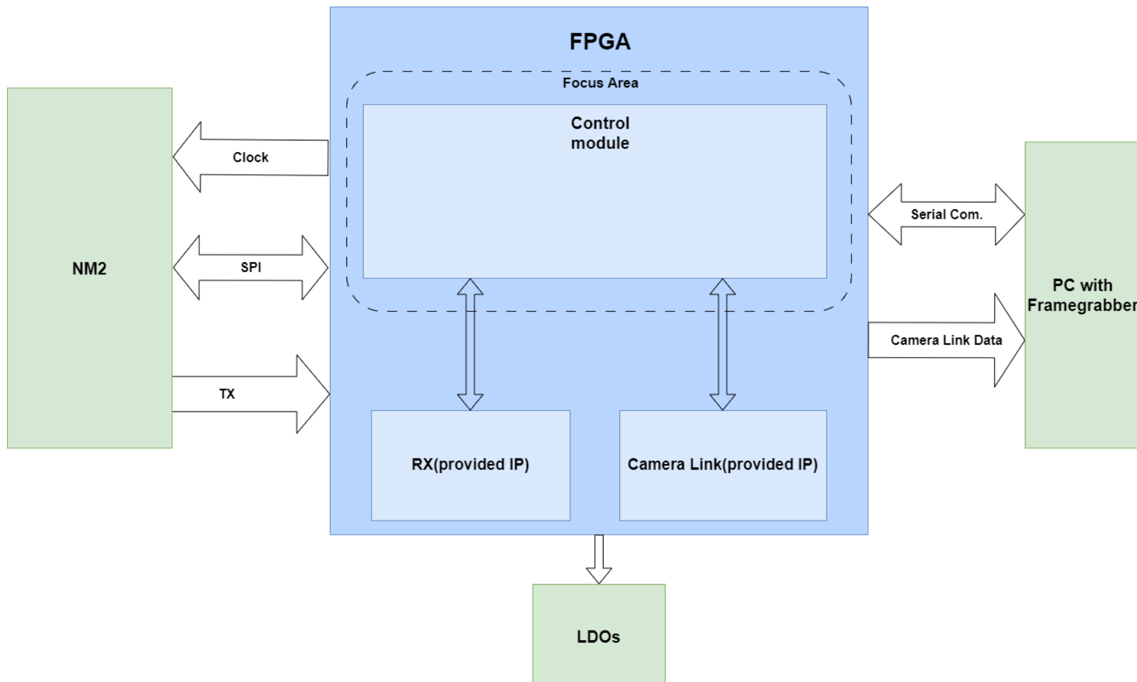


Figure 11: High level block diagram of system.

A high level block diagram showing the important parts of the system is shown in Figure 11. The figure has the NM2 ASIC on the left side connected to the FPGA on the Trenz module. The interfaces between the FPGA and NM2 are TX channels for transmitting data, an SPI interface and a reference clock. On board the NM2DB, there are several LDOs controlled by the FPGA. On the right side there is a PC

with a Matrox eV-CL Frame Grabber(Section 2.4) via the Camera Link interface on NM2DB. This thesis focuses on the control module described in Section 3.2. The control module is intended to handle serial communications through a UART interface in the Camera Link to control the NM2RX IP and Camera Link IP, as well as configuring both the NM2 ASIC and the on-board LDOs.

3.2 Control module

The control module of the firmware is depicted in Figure 12. The two IPs in the bottom, NM2RX IP and Camera Link IP, are work done previously and described in Section 2.6.1 and Section 2.5.

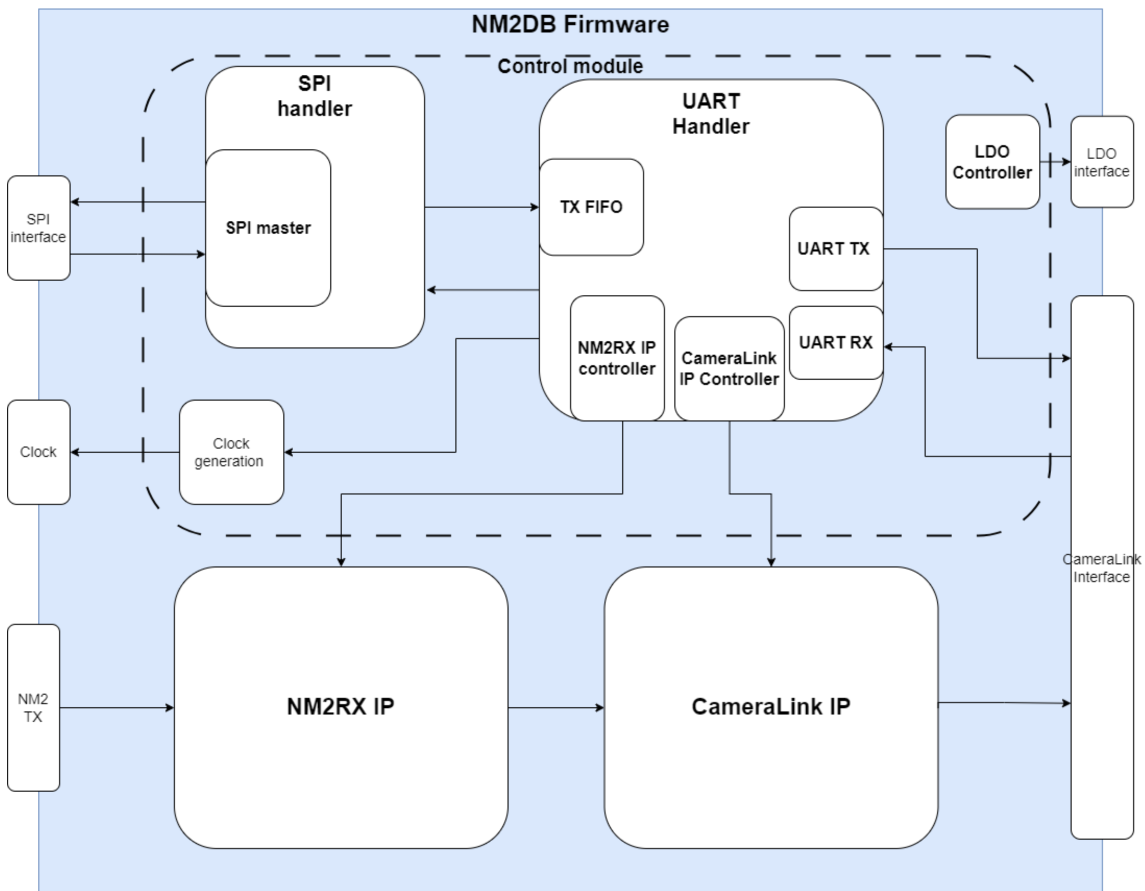


Figure 12: FPGA firmware block diagram. The control module is marked with stapled lines.

The part of the system contained in the stapled box is work done related to this project. The UART handler is a controller handling the communication with the Frame Grabber through the Camera Link serial ports. Within the UART handler, a UART RX and TX module is instantiated. When the UART RX module receives a byte it is sent to the UART handler. The UART handler then matches the byte with a map of functions and depending on which command byte is received, different functions are executed in the firmware. All the functions are explained in Section 3.5.

The SPI interface to NM2 can also be controlled by sending the appropriate command byte. If an SPI command is used the UART handler will pass it to the SPI handler which decodes it and coverts to the format accepted by the NM2 and is

transmitted by the SPI master's MOSI signal. When bytes are read on the SPI MISO signal(read commands), it will be sent to a FIFO in the UART handler. The FIFO is used in the case of reading several bytes from the SPI as the speed of the SPI interface is faster than the UART.

There is also a configurable clock provided from the FPGA to the NM2. This clock can be adjusted by sending the NM2 Clock Select command in the serial interface.

3.3 Vivado Block design

Figure 13 shows the block design created with the IP integrator. The Vivado tool has a feature called IP integrator where a high level block design is created. This tool allows you to add premade IP's, Register-Transfer Level (RTL) modules and a processor system block to configure the processor part of the Zynq SoC.

3.3.1 Configuration of Processor System

As this design is implemented fully in VHDL and not in software on the processor, the only configuration needed is the system clock provided to the Programmable Logic (PL) part of the chip. The frequency chosen for the clock is 100 MHz as this is the frequency used by the Camera Link IP.

3.3.2 Clocking Wizard

In order to easily provide different clock frequency to the NM2 ASIC, a Clocking Wizard IP from the Xilinx library is used. The Clocking Wizard takes an input clock provided by the processor system and outputs up to 8 different clocks. The clock pin of this IP is using the 100 MHz system clock used for the rest of the PL part of the design.

In the datasheet for NM2 a maximum clock frequency of 15 MHz is specified for the reference clock. The outputs from the clocking wizard are chosen as 15 MHz, 12 MHz, 10 MHz and 6 MHz. This is to both be able to run the NM2 at maximum clock frequency and be able to lower the clock rate if there is some stability issues. Outputs for the clocking wizard are connected directly to the Top module of the design as shown in Figure 14. The choice of clock frequency is done by using the serial interface and is then routed to the reference clock pins on the NM2.

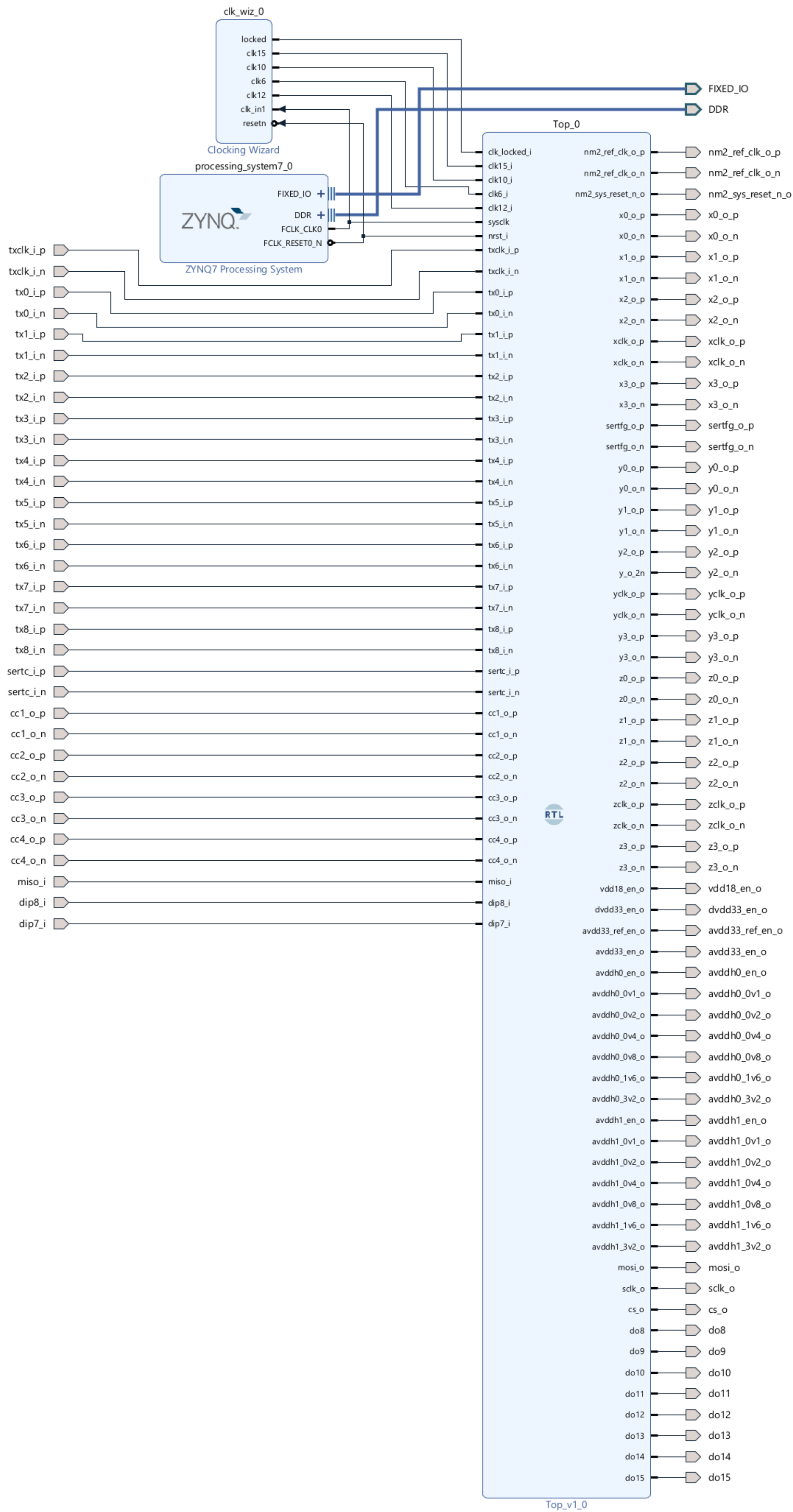


Figure 13: Block design generated in Vivado.

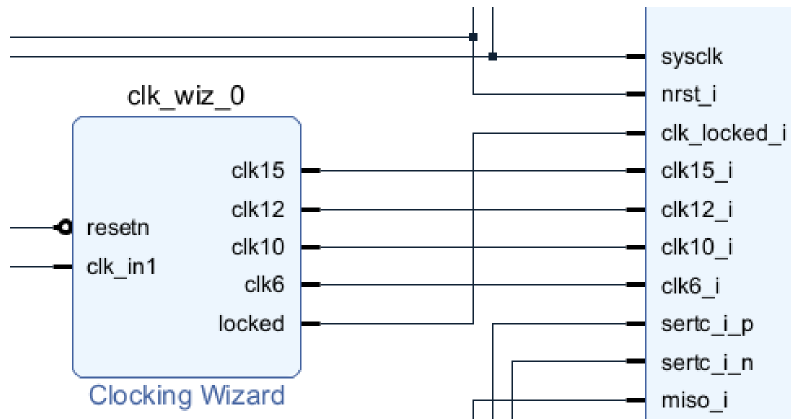


Figure 14: Clocking wizard block diagram

3.4 Configuration of Inputs and Outputs

In addition to the single-ended inputs and outputs used in the design, there is a need for both differential LVDS signaling and tri-state buffers to meet the specifications needed. Therefore the work done to configure the I/O is described here.

3.4.1 Differential buffers

Both the Camera Link channels and the upstream TX channels from NM2 are using 2.5V LVDS. To implement this in a Xilinx Zynq 7 series FPGA, two different buffers from the SelectIO guide[Xil18] are used. The differential output ports, like SerTFG in the Camera Link interface, use the OBUFDS primitive and is implemented as shown in Figure 15. This primitive takes a single ended signal, sertfg, and outputs a differential pair, in this case using the 2.5V LVDS standard.

```
--LVDS DRIVER FOR SERIAL OUTPUT TO FRAMEGRABBER
OBUFDS_sertfg : OBUFDS
generic map (
IOSTANDARD => "LVDS_25",
SLEW       => "SLOW")
port map (
O   => sertfg_o_p, -- Diff_p output
OB  => sertfg_o_n, -- Diff_n output
I   => sertfg -- Buffer input
);
```

Figure 15: Differential output buffer.

For all the differential input signals to the FPGA, the implementation is almost the same, however using a differential input buffer IBUFDS as shown in Figure 16. The important parameter to set is the DIFF_TERM to true. This terminates the input according to the $\approx 100\,\Omega$ transmission lines.

3.4.2 Tri-state signals

There are two LDOs used for external supply, AVDDH0 and AVDDH1 which are of type TPS7A4700RGWT. In the datasheet [Tex14] it is specified that the program-

```
--LVDS DRIVER FOR SERIAL INPUT FROM FRAMEGRABBER
IBUFDS_sertc : IBUFDS
generic map (
DIFF_TERM     => TRUE,
IBUF_LOW_PWR => FALSE,
IOSTANDARD    => "LVDS_25")
port map (
O  => sertc, -- Buffer output
I  => sertc_i_p, -- Diff_p buffer input
IB => sertc_i_n -- Diff_n buffer input
);
```

Figure 16: Differential input buffer.

ming of the output voltage can be done by either tying the pins to ground(enabled) or by leaving them floating(disabled). This is done in VHDL by writing the output ports as shown in Figure 17. This syntax will infer a tri-state buffer of type OBUFT from the SelectIO guide[Xil18].

```
avddh0_0v1_o <= 'Z' when avddh0_config(1) = '0' else
  '0';
avddh0_0v2_o <= 'Z' when avddh0_config(2) = '0' else
  '0';
```

Figure 17: Tri-state buffer instantiation.

3.4.3 Port mapping

In the block design described in Section 3.3, all the external ports are connected to physical pins on the Zynq SoC. The Zynq has several different banks using different voltage references and the external ports need different configurations depending on the signal type. All signals need to be constrained to the physical pins used by the NM2DB. A list of the mapping is shown in Table 6.

Table 6: FPGA port constraints.

Begin of Table			
Signal name	Bank	Pin	Pin type
nm2_sys_reset_n	33	L11N	Single ended output: 3.3V active low
nm2_ref_clk_o_p/n	34	L20P/N	Differential output: 2.5V LVDS
tx_clk_p/n	34	L13P/N	Differential input: 2.5V LVDS
tx0_p/n	34	L23P/N	Differential input: 2.5V LVDS
tx1_p/n	34	L22P/N	Differential input: 2.5V LVDS
tx2_p/n	34	L17P/N	Differential input: 2.5V LVDS
tx3_p/n	34	L9P/N	Differential input: 2.5V LVDS
tx4_p/n	34	LP15/N	Differential input: 2.5V LVDS
tx5_p/n	34	L21P/N	Differential input: 2.5V LVDS
tx6_p/n	34	L8P/N	Differential input: 2.5V LVDS
tx7_p/n	34	L10P/N	Differential input: 2.5V LVDS
tx8_p/n	34	L12P/N	Differential input: 2.5V LVDS
x0_p/n	33	L5P/N	Differential output: 2.5V LVDS

Continuation of Table 6			
Signal name	Bank	Pin	Pin type
x1_p/n	33	L6P/N	Differential output: 2.5V LVDS
x2_p/n	13	L1P/N	Differential output: 2.5V LVDS
x3_p/n	13	L14P/N	Differential output: 2.5V LVDS
xclk_p/n	13	L12P/N	Differential output: 2.5V LVDS
y0_p/n	13	L21P/N	Differential output: 2.5V LVDS
y1_p/n	13	L15P/N	Differential output: 2.5V LVDS
y2_p/n	13	L18P/N	Differential output: 2.5V LVDS
y3_p/n	13	L17P/N	Differential output: 2.5V LVDS
yclk_p/n	13	L16P/N	Differential output: 2.5V LVDS
z0_p/n	13	L20P/N	Differential output: 2.5V LVDS
z1_p/n	13	L9P/N	Differential output: 2.5V LVDS
z2_p/n	13	L11P/N	Differential output: 2.5V LVDS
z3_p/n	13	L7P/N	Differential output: 2.5V LVDS
zclk_p/n	13	L8P/N	Differential output: 2.5V LVDS
sertc_p/n	13	L13P/N	Differential output: 2.5V LVDS
sertfg_p/n	13	L4P/N	Differential output: 2.5V LVDS
cc1_p/n	13	L3P/N	Differential input: 2.5V LVDS
cc2_p/n	13	L10P/N	Differential input: 2.5V LVDS
cc3_p/n	13	L2P/N	Differential input: 2.5V LVDS
cc4_p/n	13	L23P/N	Differential input: 2.5V LVDS
avddh0_en	35	L6N	Single ended output: 3.3V
avddh0_3v2	13	L24P	Tristate buffer: Hi-z/open-drain
avddh0_1v6	13	L24N	Tristate buffer: Hi-z/open-drain
avddh0_0v8	13	L19P	Tristate buffer: Hi-z/open-drain
avddh0_0v4	13	L19N	Tristate buffer: Hi-z/open-drain
avddh0_0v2	13	L22P	Tristate buffer: Hi-z/open-drain
avddh0_0v1	13	L22N	Tristate buffer: Hi-z/open-drain
avddh1_en	35	L3P	Single ended output: 3.3V
avddh1_3v2	33	L12P	Tristate buffer: Hi-z/open-drain
avddh1_1v6	33	L12N	Tristate buffer: Hi-z/open-drain
avddh1_0v8	33	L17P	Tristate buffer: Hi-z/open-drain
avddh1_0v4	33	L17N	Tristate buffer: Hi-z/open-drain
avddh1_0v2	33	L18P	Tristate buffer: Hi-z/open-drain
avddh1_0v1	33	L18N	Tristate buffer: Hi-z/open-drain
vdd18_en	35	L1N	Single ended output: 3.3V
dvdd33_en	35	L1P	Single ended output: 3.3V
avdd33_ref_en	35	L19N	Single ended output: 3.3V
avdd33_en	35	L19P	Single ended output: 3.3V
sclk	35	L23N	Single ended output: 3.3V
cs	35	L4N	Single ended output: 3.3V
miso	35	L4P	Single ended input: 3.3V
mosi	35	L23P	Single ended output: 3.3V
dip7	34	L4P	Single ended input: 2.5V
dip7	34	L4N	Single ended input: 2.5V
End of Table			

3.5 Serial Interface

As described in Section 2.5, there are two signal pairs, SerTFG and SerTC, on the Camera Link interface dedicated to serial communication through a UART interface. Using the Camera Link interface for both data transmission and configuration or control of the system, eliminates the need for an Ethernet connector. The channels used for UART communication is Serial To Camera(SerTC) and Serial To Frame Grabber(SerTFG).

Controlling the NM2DB from the serial interface requires a data format. The format implemented is depicted in Figure 18. It is based on byte sized packets sent on the SerTC channel to the NM2DB and read responses are sent on the SerTFG channel to the frame grabber.

The serial interface is command byte oriented. The first byte of each frame is for signaling which function to use. Depending on which function, the frame is either 1, 2 or 4 bytes long.

COMMAND BYTE	SECOND BYTE	ADDRESS
SPI READ [00000000]	LENGTH[3:0]	ADDRESS[15:0]
SPI WRITE [00000001]	DATA[7:0]	
IO READ [00000010]	READ SELECT[4:0]	
SPI RESET [00000011]		
REG0 WRITE [00000100]	DATA[4:0]	
REG0 READ [00000101]		
REG1 WRITE [00000110]	DATA[4:0]	
REG1 READ [00000111]		
LDO WRITE [00001000]	LDO ENABLE[3:0]	
LDO READ [00001001]		
AVDDH0 WRITE [00001010]	AVDDH0 CONFIG[6:0]	
AVDDH0 READ [00001011]		
AVDDH1 WRITE [00001100]	AVDDH1 CONFIG[6:0]	
AVDDH1 READ [00001101]		
NM2 CLOCK SELECT [00010000]	SELECT[2:0]	
IMAGE CAPTURE [00011000]	TBD	
VIDEO CAPTURE [00100000]	TBD	

Figure 18: Serial format for NM2DB UART interface.

3.5.1 NM2 SPI commands

The first 8 functions in the serial interface starting with **00000** are all functions targeting the SPI interface on the NM2 ASIC.

SPI read and write are two functions targeting the system registers, RAM and ADC coefficient registers. These registers are all accessed through the SPI interface with a 16bit address. To read a register, SPI read(0x00) is first sent as command byte. The 4 LSB of the second byte is a length parameter and the 4 MSB are unused. The last two bytes is a 16bit address split in two bytes. To write a register, SPI write(0x01) is sent as command byte. The second byte sent is 8bit data. The last two bytes is the address.

IO read(0x02) is a function for direct read of a IRQ or DIN. The 5 LSB of read select are used to select the NM2 pin to read. The SPI reset(0x03) frame is only 1 byte. SPI reset(0x03) is a function resetting the output FIFO of NM2.

Reg0 and Reg1 read and write commands targets the two SPI registers. The write command frame is a command byte(0x04 or 0x06) followed by 5bit data where the 3 MSB are unused. To read the SPI registers, a read command is sent(0x05 or 0x07). The data returned by the UART is padded with **000** then the 5bits stored in the register.

3.5.2 LDO enable

To enable the on-board LDOs powering NM2 ASIC, a LDO write command is sent via the UART. The LDOs are enabled by sending a LDO write command and setting the corresponding bits in Figure 19 high. By writing a LDO read command, the current LDO configuration is returned. As there are only 4 bits, the configuration returned will be **0000** followed by the enabled bits.

X	X	X	X	DVDD18	DVDD33	AVDD33	AVDD33_REF

Figure 19: LDO enable data frame.

As described in Section 2.1, there are two configurable LDOs for external use. They are connected to the external pins in the external sensor interface on NM2DB which can be controlled by the AVDDH0/1 write commands. The configuration byte has a format as shown in Figure 20. Each of the two LDOs have a base output voltage of 1.4V and can be configured to output voltages up to $\sim$ input voltage $- 300mV$. The read command for these two LDOs will return the configuration as in the figure, but with a leading 0 to create a byte.

X	+3.2V	+1.6V	+0.8V	+0.4V	+0.2V	+0.1V	EN

Figure 20: External supply(AVDDH) data frame.

3.5.3 NM2 Clock Select

By using the NM2 Clock Select command, the reference clock for the NM2 ASIC can be changed. There are 5 configurations available by sending the proper byte as shown in Table 7.

UART frame	Configuration
00010 000	Clock disabled
00010 001	15 MHz
00010 010	12 MHz
00010 011	10 MHz
00010 100	6 MHz

Table 7: NM2 reference clock select.

3.5.4 Data Acquisition

Data acquisition is a feature not yet implemented into the firmware as the Camera Link and NM2RX IPs were not ready early enough to be part of this design. However, the serial interface can be extended if they are to be implemented later. Therefore the data frame for the data acquisition(image and video) are not ready yet.

3.6 Top module

As shown in the block diagram in Figure 13, there is a module called Top which is a wrapper for the modules in the system. In the top module, there are instantiations of three modules, UART handler, SPI handler and the LDO controller. All external ports are connected to the top module and routed to the appropriate modules. The internal ports between each module, e.g. the LDO config from the UART handler to the LDO controller, are interconnected with a signal in the Top module. Differential buffers for LVDS signals are also instantiated in the top module.

3.6.1 Clocks and resets

There are 3 clock domains in the firmware. These are the system clock, SPI clock(see Section 3.8.2) and NM2 reference clock. The system clock is provided by the processor system and is set to 100 MHz. This clock is used for all modules in the system. The processes in each module are triggered on either the positive edge of the system clock or a reset signal. This makes the system synchronous with an asynchronous reset. The reference clock to the NM2 ASIC is provided by the top module. The clocking wizard has 4 output clocks connected to the top module which in turn is selected through the serial interface. By default, this clock is set to 15 MHz.

There are two reset signals used in the design. Both resets are inputs from DIP switch S1 on the NM2DB close to the FPGA SoM. The first reset is on switch 8 and is used a system reset for the entire firmware. This reset sets all internal signals to a known state and sets all FSM-states to initial position. This ensures proper operation of the system when turning on. The other reset is switch 7 and is directly connected to the reset pin of the NM2 ASIC.

To properly set up the system, a sequence using the clocks and resets is recommend:

1. Turn on power to NM2DB
2. Flip pin8 of DIP switch S1(resets firmware)
3. Flip pin7 of DIP switch S1(resets NM2)
4. Run setup from Python scripts

3.7 UART

The Universal Asynchronous Receiver Transmitter(UART) standard is a fully duplex serial protocol which utilizes only two signal lines. A UART transmitter and receiver uses the same baud rate, which indicates the number of bits per second sent on a serial line. When the signal is idle, it will be logic high, or '1'. When a byte is to be sent, the signal goes low, indicating a start bit. The serial line then transmit a byte before either sending a logic high('1'), to indicate the end of transmission or a logic low('0') to indicate the start of a new byte. The data frame is illustrated in Figure 21. The width of the bits is dependent on the baud rate of the interface. The BITS_PER_CLOCK is implemented as a generic parameter in both the RX and TX modules. Calculating the number of bits per clock is done by dividing

the frequency by the baud rate. For a frequency of 100 MHz and a baud rate of 9600, the result is shown in Equation 1. In the data sheet for the Matrox frame grabber[Mat21] it is specified that the UART interface supports baud rates of 300-115200. During the initial design of the system, a baud rate of 9600 is chosen, however the CLOCKS_PER_BIT parameter can be changed to use higher baud rates.

$$\text{CLOCKS_PER_BIT} = \frac{FREQUENCY}{BAUDRATE} = \frac{100\,\text{MHz}}{9600} \approx 10417 \tag{1}$$

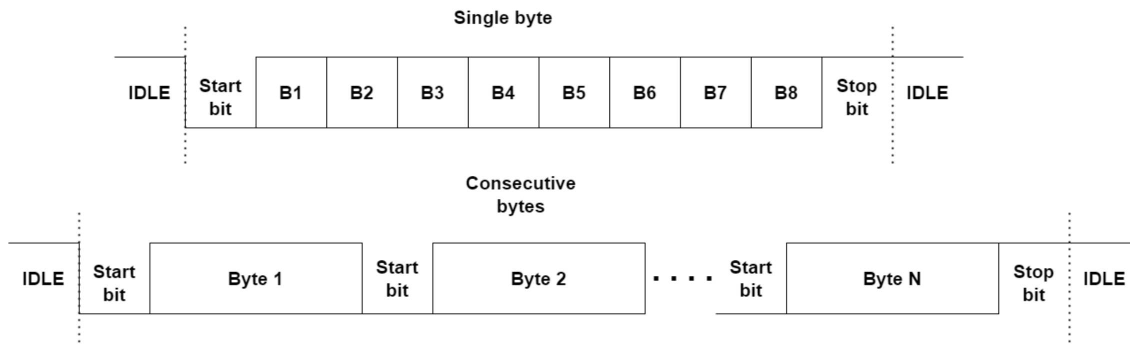


Figure 21: UART format for single and consecutive bytes.

3.7.1 UART RX

The UART RX is the module connected to the receiving channel, SerTC, of the UART interface. It is implemented with inputs and outputs as shown in Figure 22. There are three inputs, a clock, a reset and an rx_in. The rx_in is the receive signal in the UART interface. The outputs of the module are a data_byte which sends the data received, a data_valid which indicates the full byte is ready and an active signal.

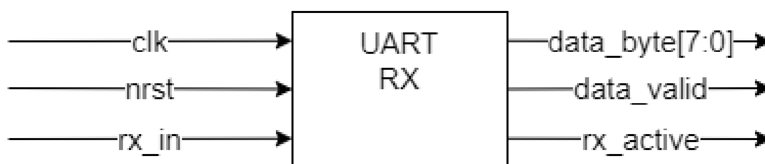


Figure 22: Block diagram for UART RX.

The UART RX module is implemented with a 4 state finite state machine(FSM) as shown in Figure 23. The FSM default state is IDLE, where the input signal is read each clock cycle. When the when rx_in goes low, the FSM switches to STARTBIT. Then the clock_counter starts counting and increments each positive edge of the clock signal. After the clock counter has counted half the clocks_per_bit parameter, we are in the middle of the bit. The FSM then switches to IDLE if the start bit was not held, or to the SAMPLE state if the start bit is still present.

In the SAMPLE state, the clock counter will be reset and count up to the number of clocks per bit. Since the counter starts in the middle of the start bit, the bit will

be read at the middle of each bit. When 8 bits(1 byte) has been read, the FSM switches to STOPBIT and will set the data_byte and the data_valid will be asserted for one clock cycle. It then waits for one more bit length before checking if the input indicates a stop bit or a new start bit. If a new start bit is detected, the FSM will read a new byte.

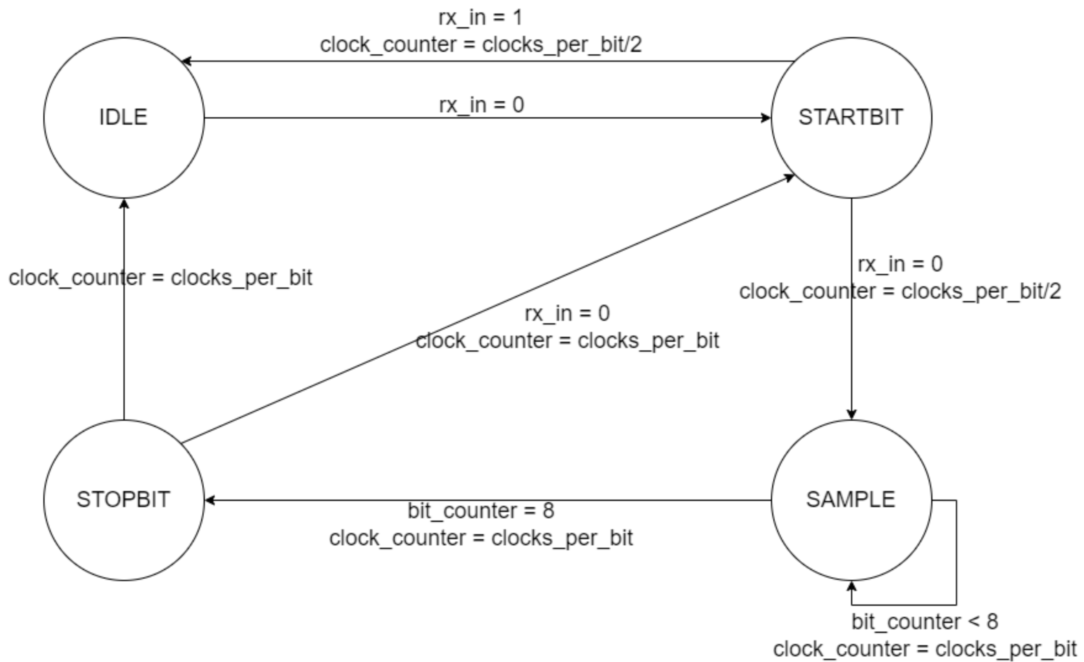


Figure 23: FSM for UART RX

3.7.2 UART handler

As shown in Figure 12, the UART handler is a controller module for 5 different modules. The UART TX and RX modules are instantiated to receive and send data through the UART interface. The TX FIFO module is a FIFO for the SPI handler to write data as the SPI operates in higher frequency than the UART. The last two modules are the controllers for the NM2RX IP and the Camera Link IP for data acquisition from the NM2 ASIC. The ports of the UART handler are shown in Figure 24.

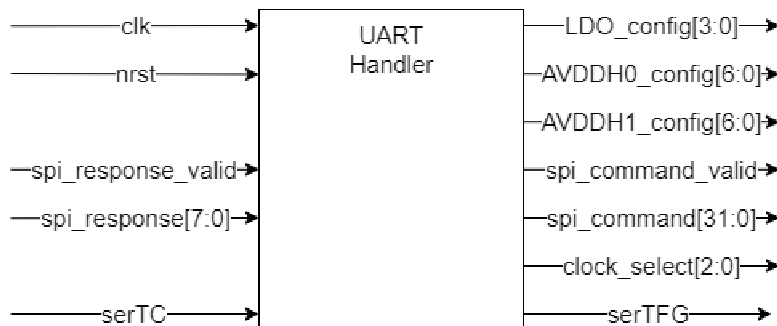


Figure 24: Block diagram for UART Handler.

The UART handler holds two FSMs to control the signals to/from the UART RX module and the UART TX module. The FSM controlling the RX module is shown

in Figure 25. The UART handler has generics corresponding to each command described in Section 3.5. When the FSM is in WAIT_COMMAND, it waits for data valid signal from the RX module. When the data valid is asserted, the data byte from the RX module is compared to the functions in the serial interface. If there is an SPI command, the FSM waits for either 2 or 4 bytes in total depending on which command is read. When the FSM returns to WAIT_COMMAND the spi_command signal is updated and the spi_command_valid signal is asserted for one clock cycle.

When sending a write command to the LDOs or the NM2 clock, the FSM reads two bytes and updates the corresponding config signals. For the LDO read signals, FSM does not change state, as there is only one byte in the data frame. It instead asserts a read signal to the TX handler which sends the config on the TX port.

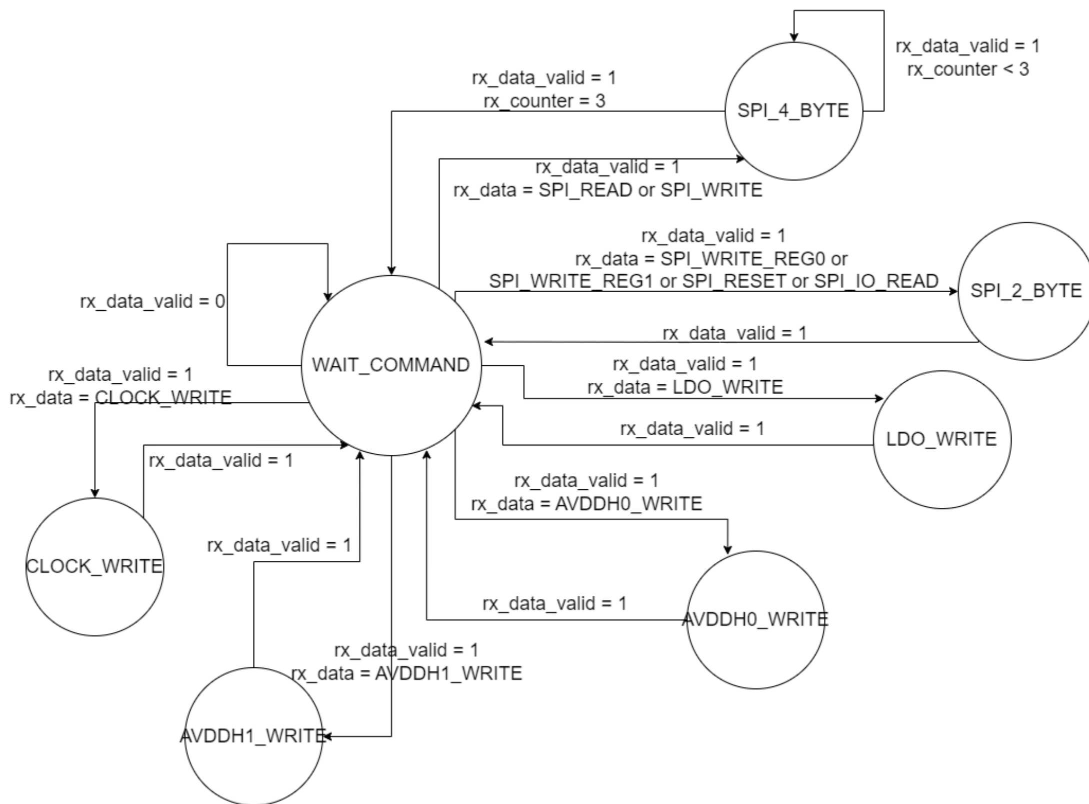


Figure 25: FSM for handling UART RX module.

The TX handler is implemented with its own FSM. In IDLE state, the FSM waits for either a read signal from the RX FSM, or to write data from the FIFO connected to the SPI handler. When the FIFO is not empty, the TX FSM will read one byte in SEND_FIFO_BYTE and then wait for a done signal from the TX module. If the FIFO is now empty, the FSM will return to IDLE, but if not, it will send another byte. In each of the states shown in the diagram in Figure 26, the enable signal for the UART TX module will be asserted for only one clock cycle to ensure that the data is transmitted only once. The three read states will return to IDLE after one clock cycle.

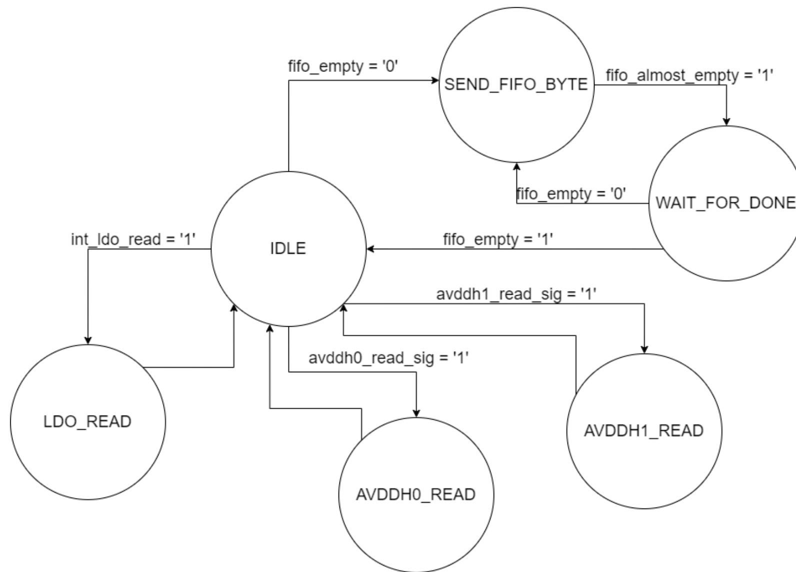


Figure 26: FSM for handling UART TX module.

3.7.3 UART TX

The UART TX module is the transmitter for the UART interface and is connected to the transmitting channel, SerTFG to the frame grabber. It has port as shown in Figure 27. The inputs of the module are a clock signal, a reset signal, an enable signal and an 8-bit data_byte. The tx_out signal is the transmit signal in the UART interface.

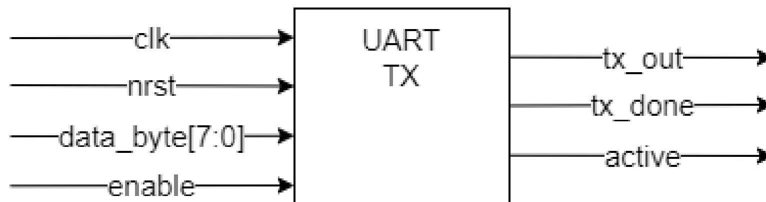


Figure 27: Block diagram for UART TX.

The FSM implemented in the TX module is shown in Figure 28. The FSM will be in IDLE until the enable signal is asserted. When the FSM is in STARTBIT, the output, tx_out, will be low to indicate a start bit. After one bit length, the FSM switches to SENDBIT state, where the output is updated every bit length. When the bit counter has iterated through all 8 bits, the done signal is asserted for one clock cycle. If the enable signal is still high, the FSM goes to STARTBIT and a start bit before sending a new byte. If it is low, it will send a stop bit before going to IDLE.

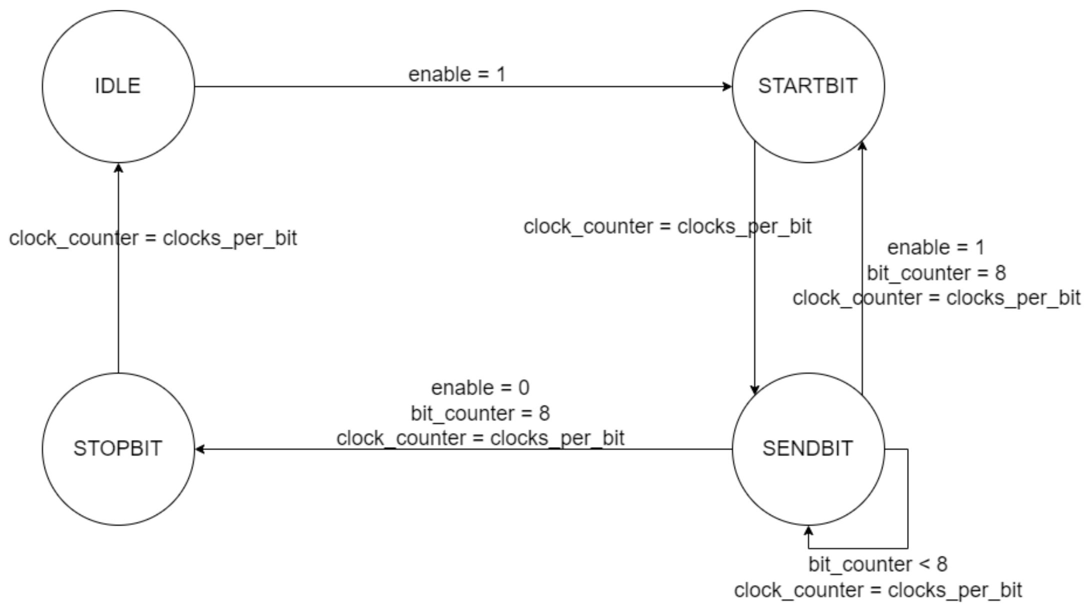


Figure 28: FSM for UART TX

3.8 SPI

The Serial Peripheral Interface(SPI) is synchronous communication protocol using four signals. The waveform for the SPI protocol is shown in Figure 29. While inactive, the Chip Select(CS) signal is held high by the master. When a new transmission is started, the CS goes low and the SPI Clock(SCLK) starts. The SPI protocol has different modes indicating whether the clock polarity(CPOL) is positive or negative and when the bits are read and shifted(CPHA). The NM2 uses a negative clock polarity meaning the clock is high in idle state. It also has CPHA = 1, meaning that bits are shifted on the falling edge and read on the rising edge of SCLK. Data from the master(controller) is sent to the slave(peripheral) on the Master In Slave Out(MOSI) signal while data from the slave to the master is sent on the Master In Slave Out(MISO) signal. The master is always controlling when data is sent by enabling the CS and SCLK meaning that the master always can write data to the slave, but the slave can not send data until the master enables CS.

3.8.1 SPI handler

The SPI handler module is the controller responsible for handling the enabling and data transmission to and from the SPI master. The SPI handler works by receiving an input command from the UART handler and sending the consecutive bytes to the SPI master. Each time a command is received from the UART, the SPI handler converts the data into the format accepted by the NM2 ASIC. As all the functions supported by the SPI interface on the NM2 are handled differently, they are all implemented as a state in the FSM implemented in the SPI handler. As shown in Figure 30, the spi data port is 32 bit wide. This port holds the command received from the UART handler as well as the data byte and addresses depending on which command is used.

When a command is received from the UART handler, it is not on the same format

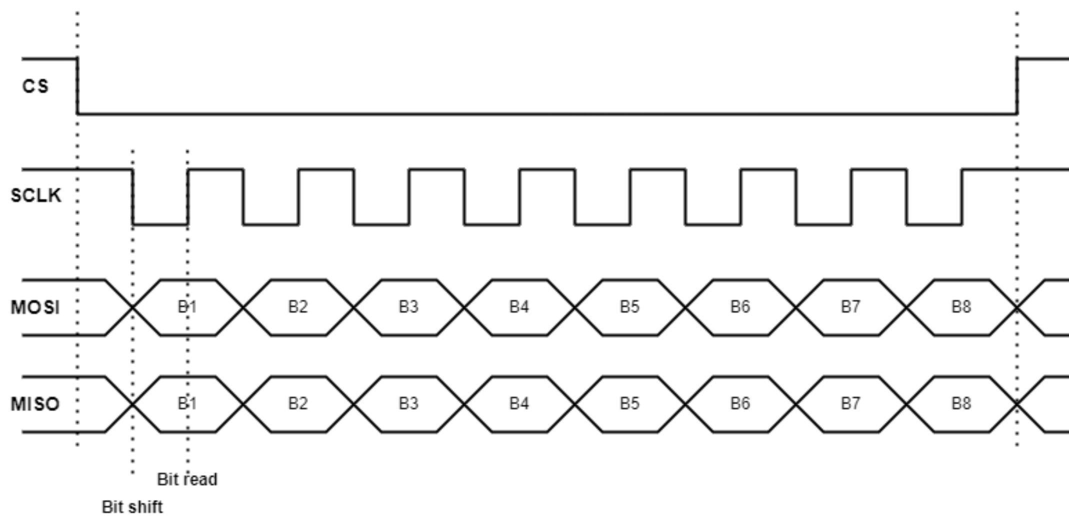


Figure 29: SPI protocol.

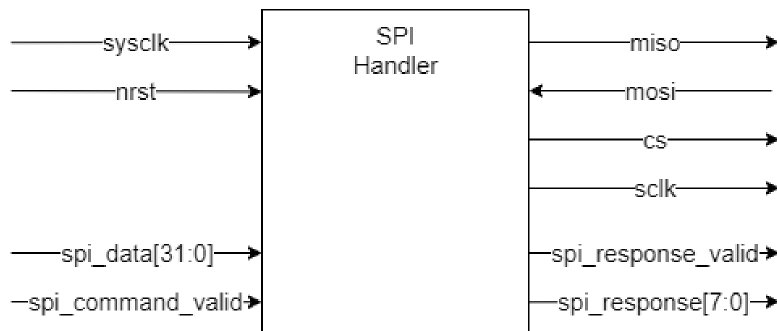


Figure 30: Block diagram for SPI handler.

as used by the NM2 SPI interface. A conversion of the data is then needed. An overview of how the data is converted is shown in Figure 31. The main difference between the two interfaces is the use of command bytes. In the UART interface, there are more functions than in the SPI interface of NM2 and has a whole byte which only specifies the function. In the SPI interface, the first 3 bits indicate which function to use, while the other bits are either used for data or as a length parameter.

One important thing to note is that the current version of the NM2 ASIC has a bug in the read interface. The bug is present while trying to use the system read function with length of more than 7 ($LLLL \geq 8$). If a length of 8 or more is passed to NM2, the read sequence on the MISO port will never start. The workaround implemented is to hard code the length parameter to always be **0001** or 1 byte. This means that only read operation of 1 byte is supported to ensure stability until the bug is fixed in the next iteration of NM2.

3.8.2 SPI master

The SPI master module is handling SPI communication according to the SPI protocol. The module is implemented with a clock speed of 10 MHz which is chosen for easy implementation with a 100 MHz system clock as the module uses a counter to toggle the SPI clock every 5 clock cycles. The inputs and outputs of the module is

Command	Format	Byte 1	Byte 2	Byte 3	Byte 4
System Read	UART format	00000 000	XXXX LLLL	Address[16bit]	
	SPI format	000 P LLLL	Address[16bit]		
System Write	UART format	00000 001	Data[8bit]	Address[16bit]	
	SPI format	001 P XXXX	Address[16bit]		Data[8bit]
IO READ	UART format	00000 010	XXX SSSSS		
	SPI format	010 SSSSS			
SPI_RESET	UART format	00000 011	XXX DDDDD		
	SPI format	011 DDDDD			
SPI_REG0_WRITE	UART format	00000 100	XXX DDDDD		
	SPI format	100 DDDDD			
SPI_REG0_READ	UART format	00000 101			
	SPI format	101 XXXXX			
SPI_REG1_WRITE	UART format	00000 110	XXX DDDDD		
	SPI format	110 DDDDD			
SPI_REG1_READ	UART format	00000 111			
	SPI format	111 XXXXX			

Figure 31: Conversion of UART to SPI data.

shown in Figure 32. The SPI master reads the send data byte when the enable is asserted. When the 8 bits are written the byte done signal goes high indicated that all bits are written. When all 8 bits on the MISO channel has been read, the data valid signal goes high for one clock cycle to indicate a new byte is read.

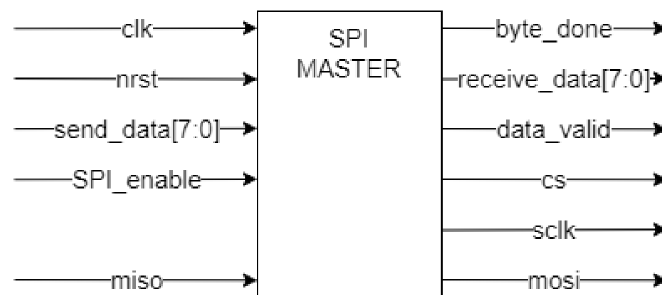


Figure 32: Block diagram for SPI master.

In the SPI master module, a FSM as shown in Figure 33 controls the SPI signal interface. When the enable signal is asserted, the FSM switches to ACTIVE where the chip select goes low and the SPI clock is enabled. The first falling edge of the SPI clock is a half period after the chip select goes low to ensure that the slave is ready with the data on the MISO signal. In the ACTIVE state, the MOSI signal is shifted out on the falling edge of the SPI clock and the MISO signal is read on the rising edge of the clock. When the SPI clock has had 8 periods, the FSM goes into the BYTE_DONE state where the data valid signal is asserted indicating that the receive data from slave is valid. If SPI enable is still high, a new active cycle starts. If the SPI enable is low, the FSM goes into IDLE and the CS will go high.

3.9 LDO controller

The 4 LDOs supplying the on-board components of NM2DB and the the AVDDH supplies for external use are all controlled by the LDO controller module. This module receives configuration registers from the UART handler and sets the appro-

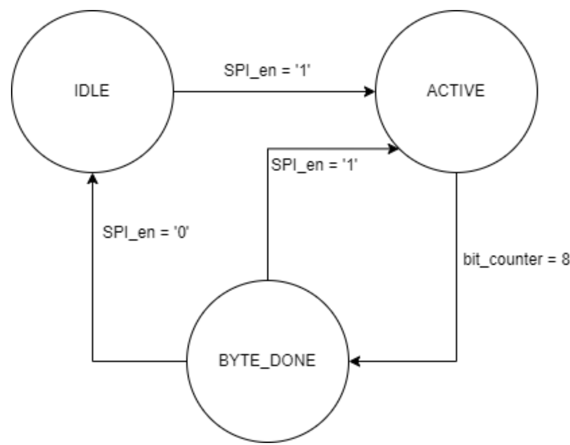


Figure 33: SPI master FSM.

priate pins on the LDOs accordingly. The 4 LDOs supplying the NM2 ASIC have enable pins which are configured as single ended outputs. These signals are within the LDO enable output ports. The two LDOs used for external supplies, AVDDH0 and AVDDH1, are both programmed with the 6 pins in the AVDDHx config ports. These ports use tri-state signals where high impedance 'Z' is disabled and ground '0' is enabled. The enable pins are single ended pins.

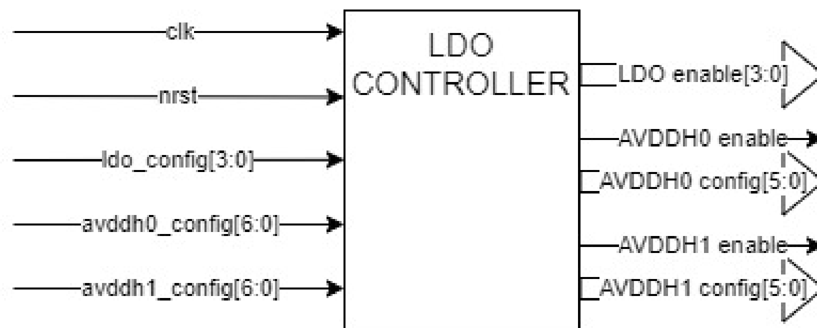
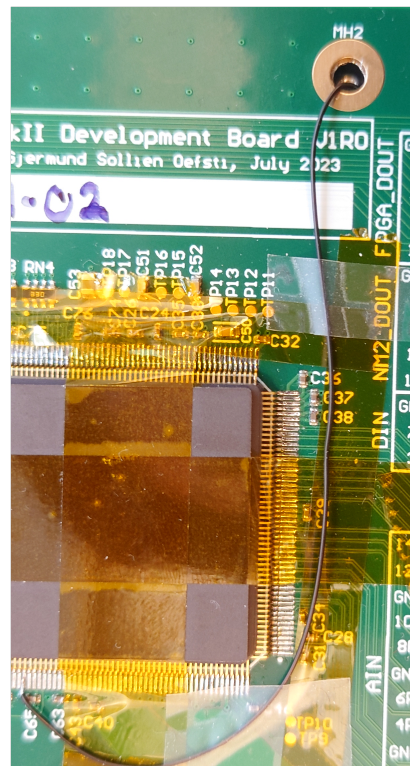


Figure 34: LDO controller block diagram.

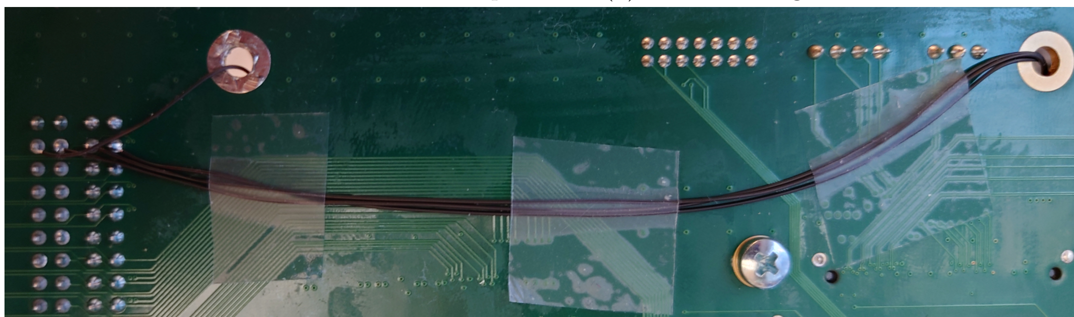
3.10 Modifications of NM2DB

Since NM2DB is made to be compatible with a previous firmware, the SPI interface ports are on the FPGA side connected to a group of pins called Multiplexed Input/Output (MIO). MIO pins are on Zynq SoCs fixed to the processor system. This means that the signals connected to MIO pins can not be accessed directly by the PL. Since this thesis is focused around VHDL and not embedded programming, a modification is made by removing resistor bank RN6(see Figure 35a) and adding wires to connecting the SPI pins to the FPGA_DOUT0-3 pins(see Figure 35c). The MISO trace which did not have a resistor had to be cut and wired from the pad for the NM2(see Figure 35b). This way, the SPI interface can be accessed by the PL.

(a) Photo showing the removal of resistor bank RN6 and wires soldered to SPI pins.



(b) Photo showing MISO wire.



(c) Photo showing back side of NM2DB with wires to FPGA_DOUT0-3.

Figure 35: Photos showing patches done to wire SPI pins to PL I/O.

3.10.1 Altered schematic

After the patches described above, the two schematics shown in Figure 36 and Figure 37 show the altered design of NM2DB. SPI signals are connected to FPGA_DOUT0-3 in connector J17 and J19. On the Trenz module's connector, the SPI signals are connected to Bank 35, pins L4 and L23.

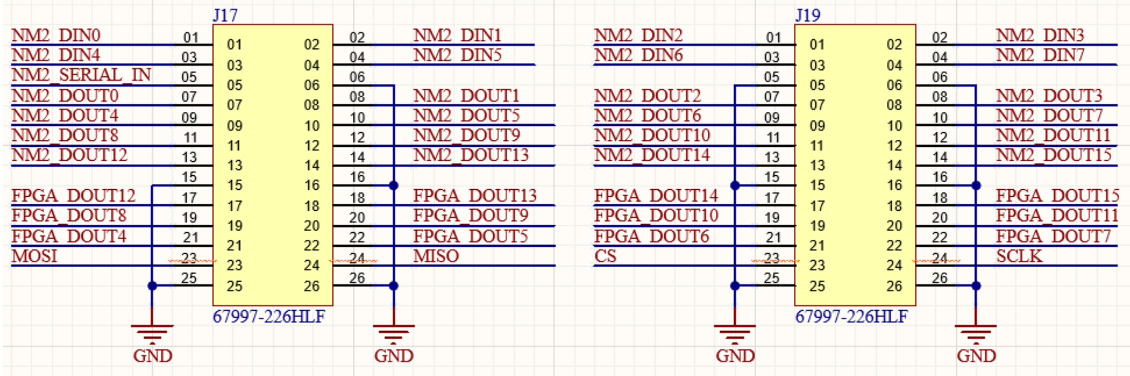


Figure 36: FPGA_DOUT 0-3 are replaced with SPI signals. This is the schematic of NM2DB after patch.

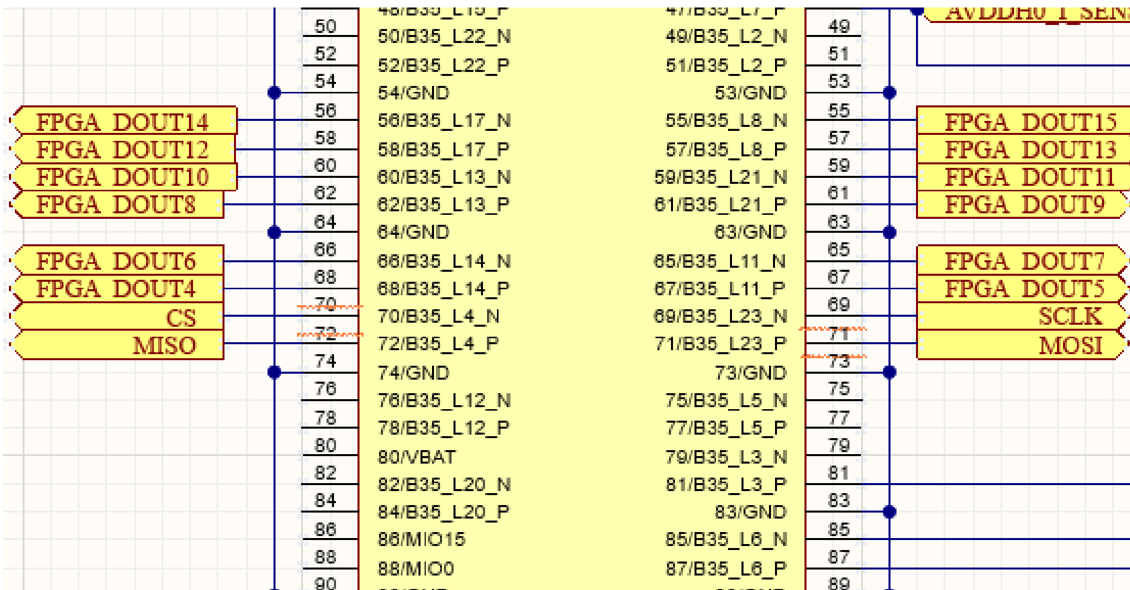


Figure 37: FPGA pins connected to SPI signals after patch. The SPI signals are connected to B35 pins L4 and L23.

4 Results

The results presented in this section are obtained from simulations in Vivado, measuring with an oscilloscope or using software on the PC with the frame grabber. Firstly the test plan is presented to give insight in which method is used for obtaining the results and where to find them. Then some results from Vivado with simulations and timing reports are presented, followed by some insight in how the system is tested and in which environment. Lastly the verification of the system is presented with some experimental testing to see how fast the implementation can be.

4.1 Test Overview

An overview of the tests is provided in Table 8. The table contains an overview of the result section and the method of how the results are obtained.

Test	Method	Section
Pre-synthesis verification	Simulation of all firmware functions in Vivado to verify the correct operation. A test bench for each sub-module is written and a test bench for the top module. The simulation wave forms are inspected to check that modules behave as expected.	Section 4.2
Post-implementation timing and utilization	Verification of the timings in the system. The timing report is inspected to verify that timing constraints are met. Also post-implementation utilization is presented.	Section 4.3
UART verification	Verification of the UART implementation. The UART interface is tested to verify that all functions operate correctly.	Section 4.6
LDO control verification	Verification of the LDO enabling and programming of AVDDH.	Section 4.6.2
SPI verification	Verification of the SPI implementation. The SPI signals are verified to meet the standard and the SPI functions implemented are tested towards NM2.	Section 4.7
NIRCA MkII control	Test reading and writing to all memory spaces on NM2 through Python scripts on PC with frame grabber.	Section 4.8
Performance testing	Test limits of system with higher baud rates.	Section 4.9

Table 8: Test overview for firmware.

4.2 VHDL module simulations

While implementing the design in Vivado, all VHDL modules have been tested by writing a test bench and verifying that the module behaves as expected before implementing it in the top module. The simulations for each module are placed in appendix A. In Figure 38 the waveform for the SPI master test bench is shown as an example.

The waveform is produced by first applying a reset signal and then passing a byte on the send data port. The expected behaviour of the module is to activate the SCLK

signal and shift out one bit on MOSI each falling edge. On the MISO signal, there should be a new byte read every 8 clock periods. The module behaves as expected as the MOSI signal matches the data byte(0x5F) by looking at the signals behaviour each falling edge. CS is low the whole transmission and the input signal MISO is read every rising edge as seen in the receive_data_sig.

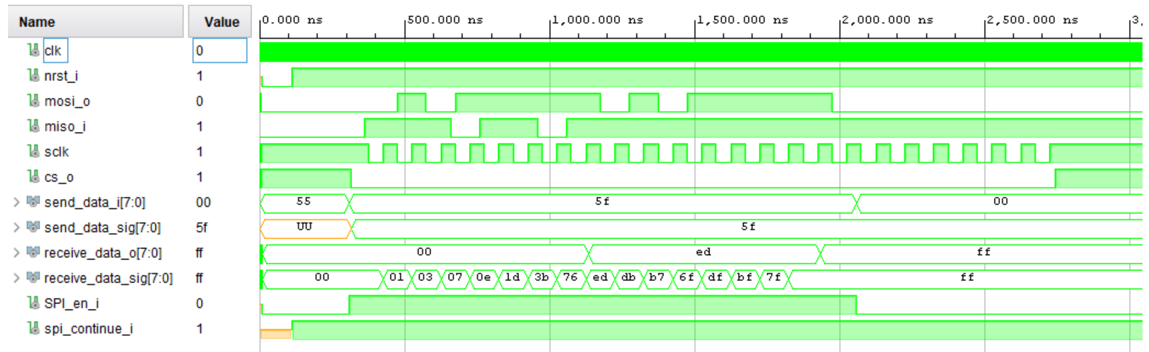


Figure 38: SPI master simulation.

For the rest of the modules, the same procedure for testing in Vivado is done, but not explained in detail here. However, to simulate the behaviour of the entire system, the top module is simulated with all it's functions before testing on the actual FPGA. This is done by only applying stimuli to the external ports(those accessible in the external connectors on the NM2DB). In Figure 39, a test enabling the 3 of the 4 on-board LDOs using the LDO write command(**0x08** + **0x0B**) is shown. The waveform show VDD18, DVDD33 and AVDD33 enabled and AVDD33_REF disabled as expected.

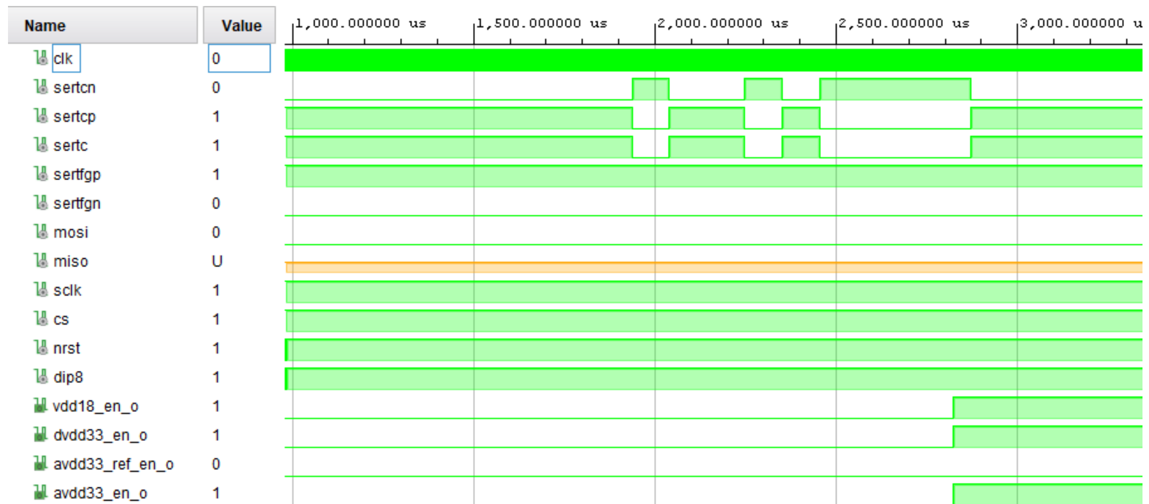


Figure 39: Top module simulation, LDO enable.

For the other pre-synthesis simulation tests, the results are shown in Table 9. This table shown which function is tested and what stimuli is impressed to the UART ports. The results are obtained by looking at the waveform.

Test function	UART stimuli	Result
LDO Write	**00001000** + **XXXXEEEE**	Ok: The correct LDO pins according to **EEEE** are toggled to high.
LDO read	**00001001**	Ok: The current LDO config is written on UART tx(SerTFG).
AVDDH0 Write	**00001010** + **XTTTTTTE**	Ok: LDO is enabled, enabled pins are '0' and disabled pins are 'Z'.
AVDDH0 read	**00001011**	Ok: The current AVDDH0 config is written on UART tx(SerTFG).
AVDDH1 Write	**00001100** + **XTTTTTTE**	Ok: LDO is enabled, enabled pins are '0' and disabled pins are 'Z'.
AVDDH1 read	**00001101**	Ok: The current AVDDH1 config is written on UART tx(SerTFG).
WRITE REG0	**00000100** + **XXXDDDDD**	Ok: SPI interface is sending **100DDDDD**.
READ REG0	**00000101**	Ok: **000DDDDD** is sent on UART tx(SerTFG).
WRITE REG1	**00000110** + **XXXDDDDD**	Ok: SPI interface is sending **110DDDDD**.
READ REG1	**00000111**	Ok: **000DDDDD** is sent on UART tx(SerTFG).
SPI READ	**00000000** + **0000LLLL** + Address[16bit]	Ok: SPI interface sends the whole frame. SPI master reads requested number of bytes and sends them to FIFO. The FIFO bytes is ent on UART tx.
SPI WRITE	**00000001** + data[8bit] + Address[16bit]	Ok: SPI interface sends the whole frame.
SPI IO READ	**00000010** + **XXXXSSSS**	Ok: SPI interface sends command and stays enabled until a new IO read is sent.
SPI RESET	**00000011**	Ok: SPI interface sends **01110110** then **01110000**.

Table 9: Simulation results from Vivado. E = Enable bit, T = Toggle bit, D = Data bit, L = Length bit.

4.3 Vivado Timing and Resource Utilization

After running the implementation is Vivado, a timing summary is generated to detect any possible problems. In Figure 40 the timing summary is shown. We see that all three parameters are blue, which means they all meet the timing requirements and no errors are likely to occur due to timing.

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	4.744 ns	Worst Hold Slack (WHS):	0.121 ns	Worst Pulse Width Slack (WPWS):	3.000 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	797	Total Number of Endpoints:	797	Total Number of Endpoints:	456

All user specified timing constraints are met.

Figure 40: Post-implementation timing summary.

For the utilization of FPGA resources, a summary is given in Figure 41. This is the utilization without adding NM2RX and Camera Link IPs. After adding those, the LUT and FF utilization would increase a lot as these are more complex modules. The IO would also increase a big amount as all Camera Link and TX signals are not constrained in the current version.

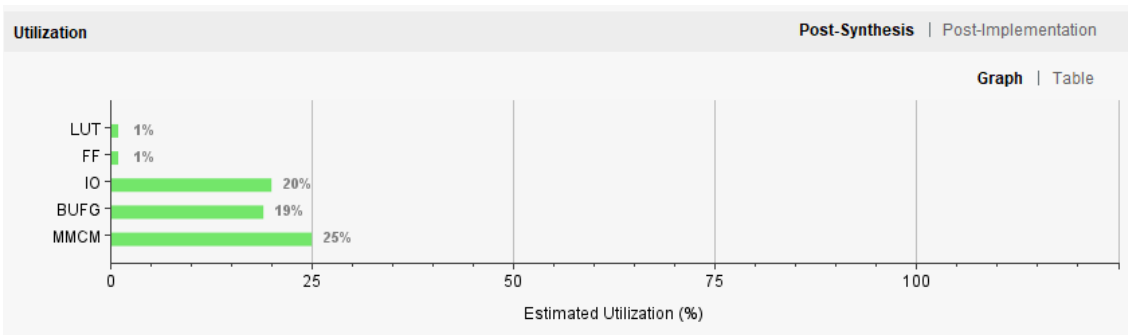


Figure 41: Post-implementation utilization. NM2RX and Camera Link IPs are not implemented and their signals not constrained.

4.4 Test setup

The test setup used to test the system is shown in Figure 42. A power supply providing 5V DC in the bottom left is connected to the power input of NM2DB. To monitor and measure the signals on board, a oscilloscope in the top is used. The oscilloscope has in addition to analog inputs, a digital decoder with the blue ribbon cable to monitor digital signals.

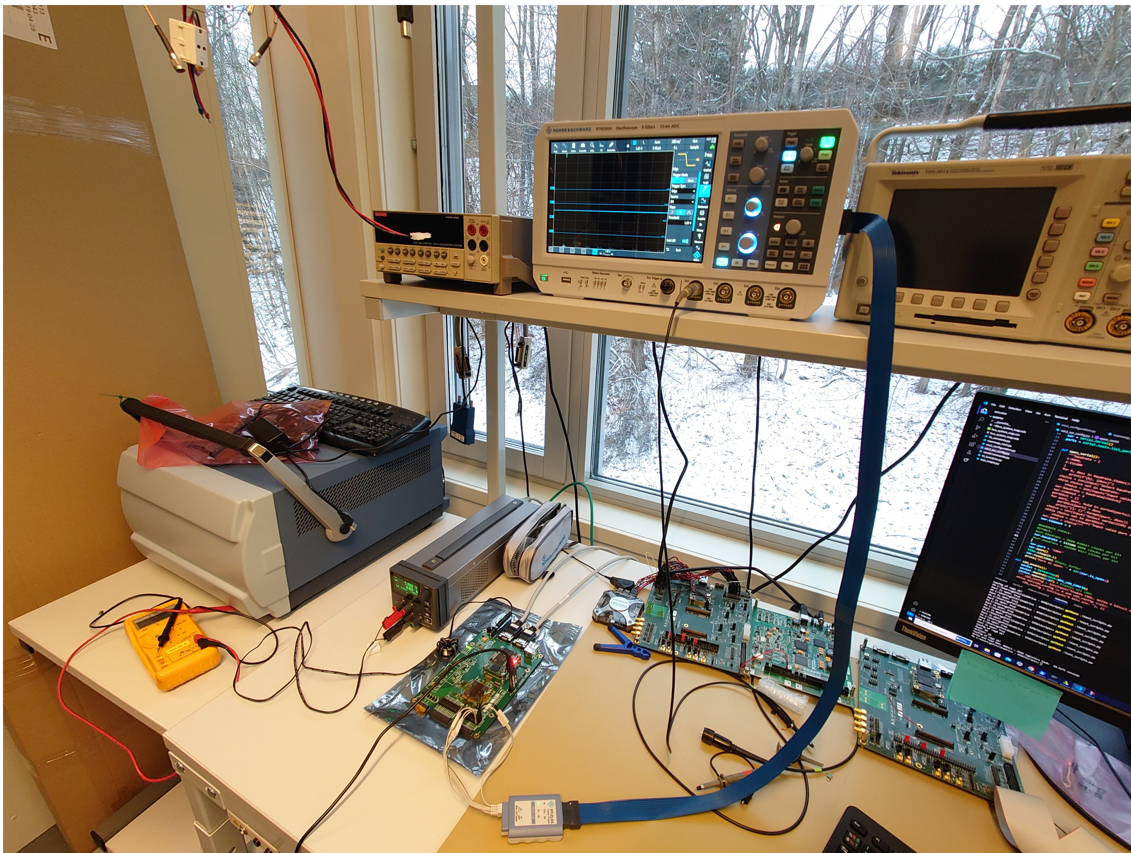


Figure 42: Picture of lab test setup. NM2DB is connected to the PC with Frame Grabber with Camera Link Cables and a JTAG cable. The blue ribbon cable is used to measure digital signals. A power supply is connected to the power input.

As seen in Figure 43, there are two Camera Link connectors connected to the left side of the PCB. The FPGA is programmed using a JTAG connector in the top

middle. The JTAG is recognized automatically by Vivado. To monitor the SPI interface, 4 signal wires from the oscilloscope are connected to the FPGA_DOUT0-3 pins which are modified as explained in Section 3.10.

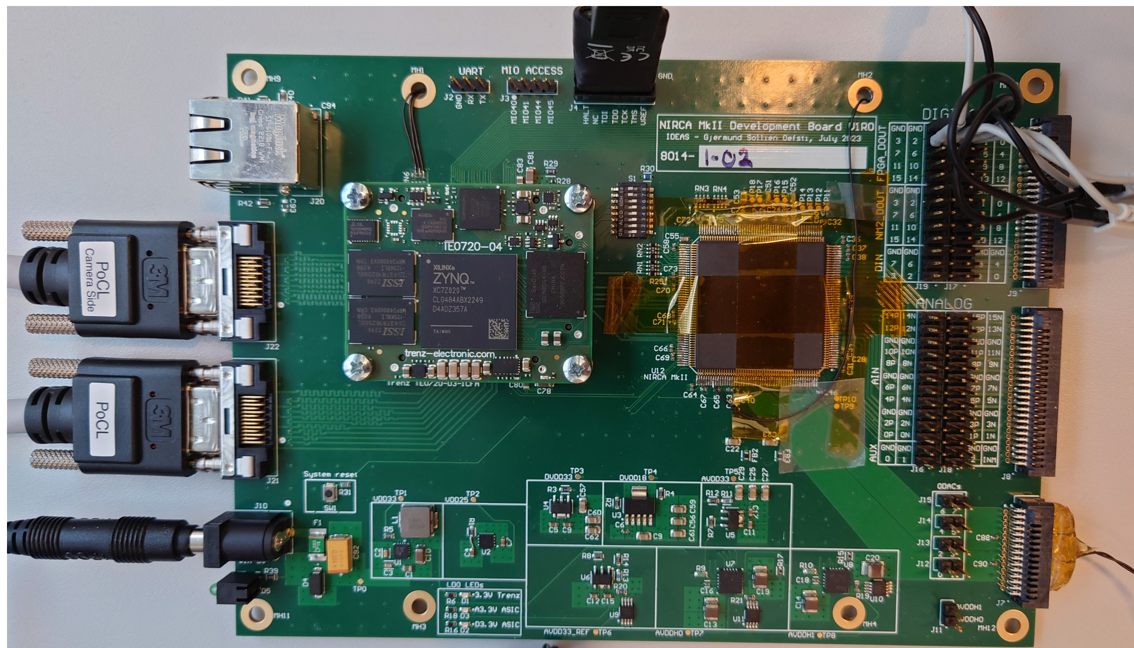


Figure 43: Picture of NM2DB with connectors. Left side: two Camera Link connectors and a power supply. Top: JTAG. Right side: Digital probes.

4.5 PySerial

While most of the pins in the Camera Link interface are used by the Frame Grabber and need to use the MIL library provided by Matrox, the two serial ports, SerTFG and SerTC used in this thesis are available in windows as a COM port accessible by the operating system(in this case Windows). To use this port, a Python library called PySerial[Lie23] is used both to write and read from the UART interface.

PySerial is a library which gives an interface to the Com ports connected to the PC. To use the PySerial library there are some parameters which need to be configured correctly before opening the port. An example configuration is shown in Figure 44. COM5 is the serial port listed in the device manager in Windows for the Matrox frame grabber. The baud rate matches the one implemented in the FPGA firmware. This parameter is changed during testing.

```
import serial
ser = serial.Serial()
ser.baudrate = 9600 #10417 clocks per bit
ser.port = 'COM5'
ser.open()
```

Figure 44: PySerial example configuration.

To send data bytes over the PySerial interface, the data is packed as a byte array. An example write operation is shown in Figure 45. This write operation enables all

LDOs by first writing command byte 0x04 and then 0x0F.

```
packet = bytearray()
packet.append(const.NM2_LDO_CONFIG)
packet.append(0x0f) # Enables all LDOs
ser.write(packet)
```

Figure 45: PySerial example write.

On the receiving channel, a read operation can be done by using the read() function in PySerial. This function reads a specified number of bytes in the input buffer or until a timeout is reached.

In appendix B, all python functions used to obtain the results in this section are listed.

4.6 UART verification

Two main considerations were made when implementing the UART interface, speed and stability. The speed, in this case the baud rate, should be as high as possible without compromising the stability of the system. Initially, before any testing, the UART interface was implemented with a baud rate of 9600 to be on the safe side in terms of stability as 9600 is regarded as a somewhat slow baud rate.

4.6.1 Verifying the UART protocol

Before testing any of the on-board functions in the serial interface, the implementation of the UART interface was tested and measured using the oscilloscope. This is to test and verify the functionality of the UART RX and TX modules and that they comply with the UART protocol. One way to test a simple write and read operation is to send a LDO read command. The RX module then needs to read the incoming byte and if that byte matches the correct pattern, the TX module should send the current LDO config back to the frame grabber.

An example read operation is shown in the waveform in Figure 46 and is retrieved by writing a LDO read command to the UART interface and by probing the SerTFG(TX) signal. The image shows a start bit where the signal goes low. It then transmits bits for $\approx$ 950 µs before going idle. As the baud rate is 9600, the expected length of a transmission is 9 bit/9600 baud = 938 µs which is what is shown in the waveform. In the PySerial interface, the read command returns 0x03(b00000011). This read operation then proves that a single command can be executed and that a single byte can be read.

From the waveform in Figure 46 the LVDS characterisation can also be verified. We can see that the signal is centered around 1.25 V and that the amplitude is around 450 mV. As this is the positive channel of the differential pair, it is high when idle. This behaviour is within the LVDS standard.

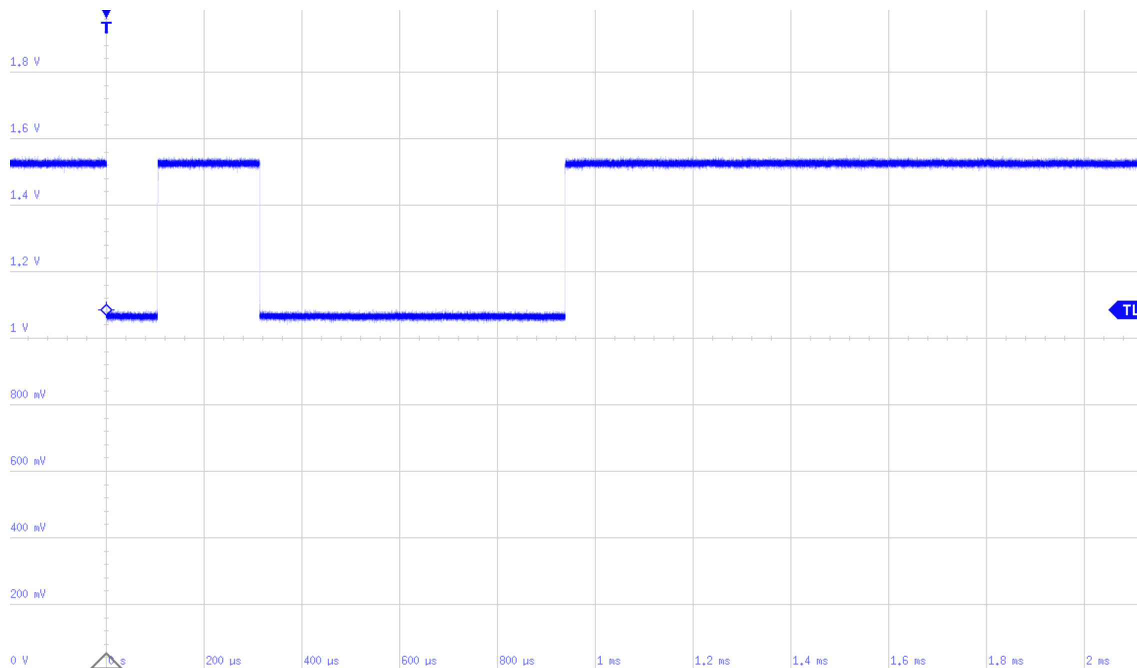


Figure 46: Measurement of SerTFG's positive line. Read response by sending LDO read command(0x09 LSB first).

4.6.2 LDO configuration write and read

To further test the UART implementation, LDO write and read functions are tested. The LDO write function should enable and disable the LDOs. As there are 4 on-board LDOs, the configuration of these ranges from 0x00 to 0x0F. 3 of the LDOs have an LED to indicate if they are on or off, while the LDO supplying DVDD18 does not. Visual inspection can therefore be done by checking the toggling of the LEDs and by measuring the voltage of the LDOs with a oscilloscope or a multi meter. The testing of the LDOs are therefore done as shown in Table 10. The last test in used by the test snippet in Figure 47. From the results, the LDOs behave as expected.

```
for i in range(0x0F):
    ser_func.LDO_enable(i)
    readValue = ser_func.LDO_read()
    if(readValue != i):
        print("LDO read value does not match the one written.")
```

Figure 47: Python snippet for testing all LDO enable configurations.

4.6.3 AVDDH programming

AVDDH0 and AVDDH1 are both programmed in the same way, by sending a AVDDH0/1 write comannd(**0x0A**/**0x0C**) followed by a config byte as described in Section 3.5. To test the implementation of the programming of the two LDOs, a test script as shown in Figure 48 is used. The script enables the LDO after 3 seconds and then iterates through all the possible configurations from **0x00** to **0x3F**.

Test description	Result
Enable only DVDD18	Measured 1.8V on test point.
Enable only DVDD33	LED lit. Measured 3.3V on test point.
Enable only AVDD33	LED lit. Measured 3.4V on test point.
Enable only AVDD33_REF	LED lit. Measured 3.4V on test point.
Write all configurations from 0x00 to 0x0F and read the configuration between each write operation.	Read data matches the written configuration.

Table 10: LDO control test procedure.

```
ser_func.AVDDH0_write(0x00)
time.sleep(3)
ser_func.AVDDH0_write(0x01)
for i in range(0x3f):
    time.sleep(.2)
    ser_func.AVDDH0_write(i*2+1)
    readValue = ser_func.AVDDH0_read()
    if(readValue != i*2 + 1):
        print("LDO read value does not match the one written.")
```

Figure 48: Python snippet for testing all AVDDH0 configurations.

After running the script, the output of the AVDDH0 LDO was measured with the oscilloscope. The result show that when turned off, the output voltage is about 0.5 V. When enabled, the first value is 1.4 V as specified in the data sheet[Tex14]. Further on, the increase for each step seems to be linear, which indicates that the output from the FPGA is configured correctly. Maximum output voltage is measured to be 4.4 V. Increasing the voltage after this point is not possible.

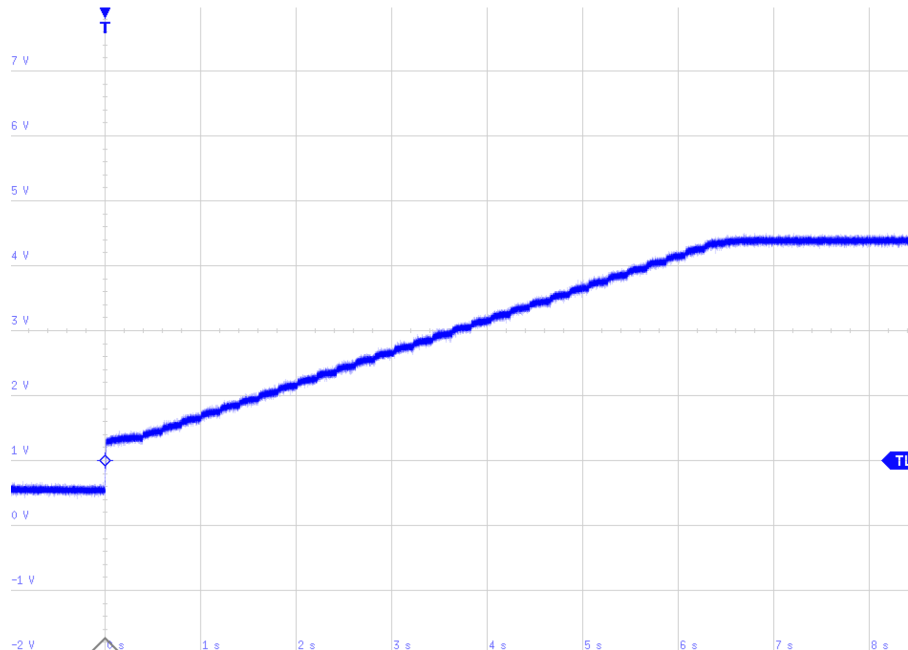


Figure 49: Test configuration of AVDDH0. LDO is enabled at 0 s and incremented by 0.1 V every 0.2 s

4.6.4 NM2 Clock Select

To verify that all clocks routed from the Clocking wizard to Top module can be selected as the reference clock to NM2, a clock select command is sent to the serial interface. A visual inspection is done by measuring the positive diff signal of the reference clock pin. In Figure 50, a measurement is shown after selecting the 6MHz clock. Looking at the measurement the frequency output can be calculated $\frac{6periods}{1\,\mu s} = 6\,\text{MHz}$. The rest of the clock selects are tested the same way with results as shown in Table 11 which shows all clocks works as expected.

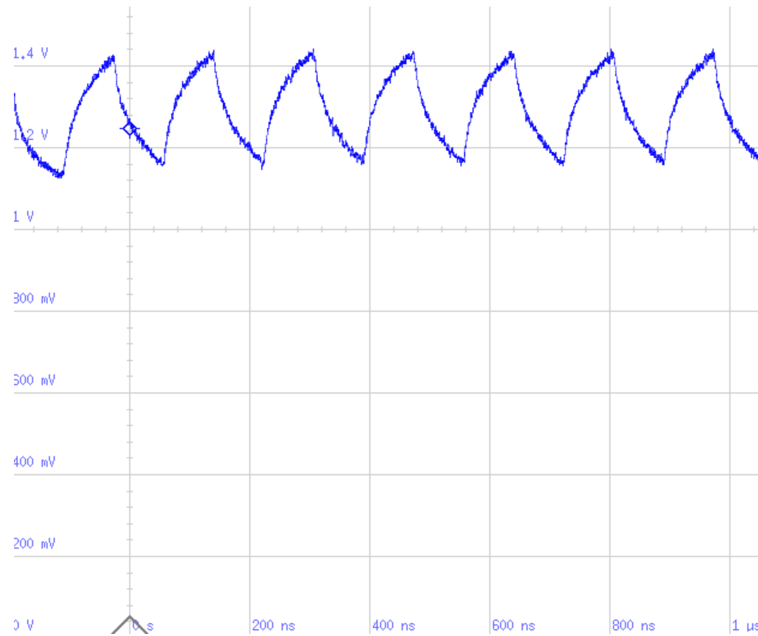


Figure 50: 6 MHz NM2 reference clock. Positive differential signal.

Clock select	Result
Disable(0x00)	Output frequency: 0MHz, DC signal.
15MHz(0x01)	Output frequency: 15MHz
12MHz(0x02)	Output frequency: 12MHz
10MHz(0x03)	Output frequency: 10MHz
6MHz(0x04)	Output frequency: 6MHz

Table 11: NM2 reference clock selection.

4.7 SPI verification

The SPI interface is connected directly to the SPI ports on the NIRCA MkII ASIC and needs to meet the requirements of NM2 to both be able to write data and read the correct data from the ASIC. This means that the SPI protocol must be implemented correctly in the SPI master as well as the SPI handler which needs to control the data sent and received. One of the simplest commands to test is a SPI register write function to the NM2. The command is dependent on the UART handler to receive and pass the command to the SPI handler. In Section 4.6 it is shown that the UART interface can receive bytes. Here the UART and the SPI is tested in conjunction.

To obtain the results in Figure 51, a Reg1 write(**0x06**) command was sent through the UART RX channel(SerTC) followed by **0x12**. When starting a write, CS goes low and the clock starts counting after a half SPI clock cycle. The clock has negative polarity as it starts high when idle. Data bits are shifted on the falling edge of the clock. The frequency of the SPI clock is derived by taking a rough estimate of the length of 8 clock periods: $\frac{8}{\approx 800ns} \approx 10\,\mathrm{MHz}$. The MOSI channel can be read by looking at what data is present at the rising edge of the SPI clock and it shows **11010010**. **110** is the SPI write reg1 command and **10010** is the data bits sent to the UART. By the above, it seems that the SPI module is capable of writing a full byte to the NM2 ASIC.

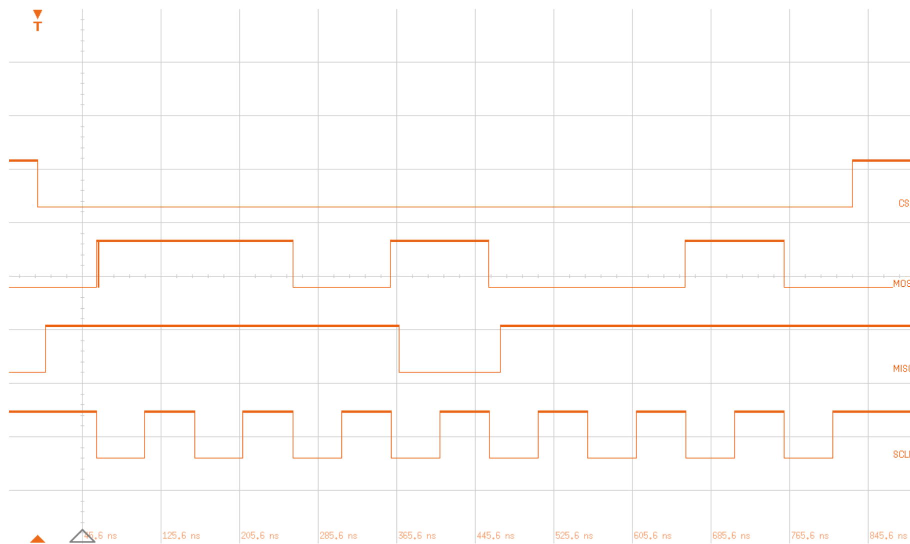


Figure 51: Write Reg1 using SPI. Data written is Command(**110**) + Data(**10010**).

Further testing of the interface is done by reading from the same register, SPI register 0. This is done by sending the SPI read reg0(**0x07**) command to the UART. The result is shown in Figure 52. From the measurement, the MOSI channel start with SPI read reg1(**111**) + **00000** which are ignored by NM2. On the MISO channel, the transmission starts with a 0-byte followed by **0x12** twice before going idle. This is also confirmed when receiving the data on the UART TX channel(SerTFG). One thing to note is the fact that CS goes high only after the byte is sent twice on the MISO channel. This is not intended and probably a bug from the SPI handler where it disables the SPI master too late. This is however not affecting the functionality of the system neither the performance as the bottleneck is still the UART interface running much slower than the SPI.

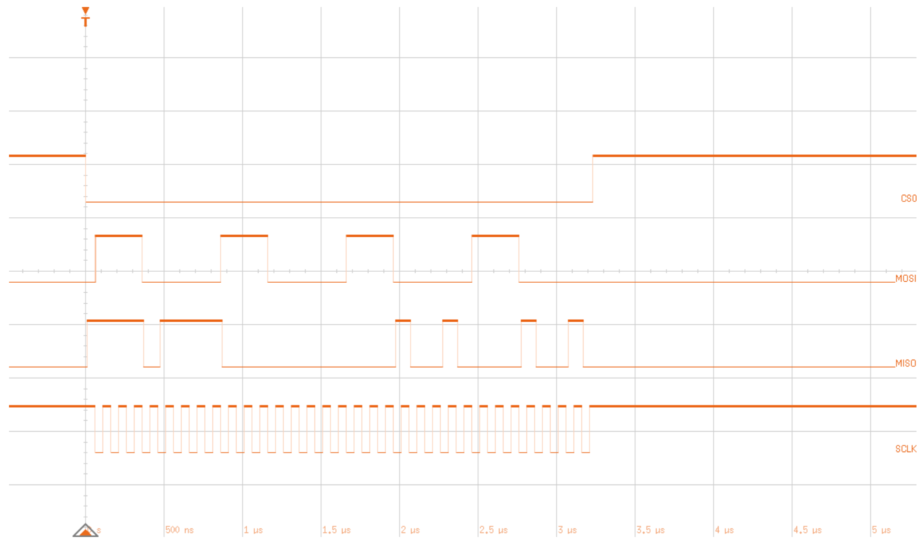


Figure 52: Read Reg1 SPI test. Data written is Command(**111**)+ignored bytes **00000**. Data received is **0x12**

4.8 NIRCA MkII ASIC control

As the SPI signals are correctly working, the interfacing with NM2 needs to be tested. To gain access to the system bus in the NM2, the two SPI registers, reg0 and reg1, need to be configured correctly. The default configuration for reg0 used here is shown in Table 12 or in hex: 0x05. For register reg1 the default configuration used is in Table 13 or 0x12 in hex.

Bit	Name	Value
2	seq_halt	1
1	seq_reset	0
0	sys_clk_enable	1

Table 12: Default reg0 setup.

Bit	Name	Value
4	pll_enable	1
2:3	seq_reset	00
0:1	clk_div_mode	10

Table 13: Default SPI reg1 setup.

After restarting NM2DB(either by turning power on or by programming the FPGA) a initial setup is done:

1. Reset dip switch 8(system reset)
2. Reset dip switch 7(NM2 reset)
3. Enable all LDOs supplying NM2
4. Set NM2 reference clock to 15 MHz
5. Run initial NM2 setup(reg0 = 0x05, reg1 = 0x12)

The initial setup is done in python by using the functions shown in Figure 53. The function NM2_init_setup() writes 0x05 to reg0 and 0x12 to reg1 and then reads both registers to verify that they are correctly written.

```
ser_func.open_serial()
ser_func.LDO_enable(0x0F)#All LDOs "1111"
ser_func.NM2_clock_select(1)#15MHz
ser_func.NM2_init_setup()
```

Figure 53: Setup procedure for serial interface.

After setting up reg0 and reg1, the SPI should now have access to the system bus. To write and read to specific registers on NM2, the memory map developed by IDEAS was partly utilized while also writing a new hook function targeting the new system and replacing a previous hook function which utilizes the TCP port. An example on how the memory map is used, is shown in Figure 54. The write value function operates on a memory field object containing the address and current value. Inputting the value to be written and the hook, in this case NM2_write, writes the value to the register address in the memory map.

```
def ODAC0_write(value):
    MM_CONF.fields['odac0_dac_lo'].write_value(value%256, hook = NM2_write)
    MM_CONF.fields['odac0_dac_hi'].write_value(int(value/256), hook = NM2_write)
```

Figure 54: Write ODAC0 register using memory map.

4.8.1 Write register test

To verify that registers are in fact written correctly and that the NM2 ASIC operates correctly, a visual verification is done by writing to the ODAC registers on NM2. All ODACs have 10bit configuration, which means the lower 8 bits are in a registers called 'odacx_dac_lo' and the 2 higher bits are the two LSBs in 'odacx_dac_hi'. Using the script shown in Figure 55, ODAC0 and ODAC2 are enabled then ODAC0 is incremented while ODAC2 is decremented.

```
ser_func.ODAC_enable()
for i in range(0x3ff):
    time.sleep(0.01)
    ser_func.ODAC0_write(i)
    ser_func.ODAC2_write(0x3ff-i)
```

Figure 55: Python loop for testing ODAC0 and ODAC2.

By running the python code in Figure 55 and measuring the outputs using the oscilloscope, the result is shown in Figure 56. This result show that registers in configuration memory can be written and that the NM2 operates correctly.

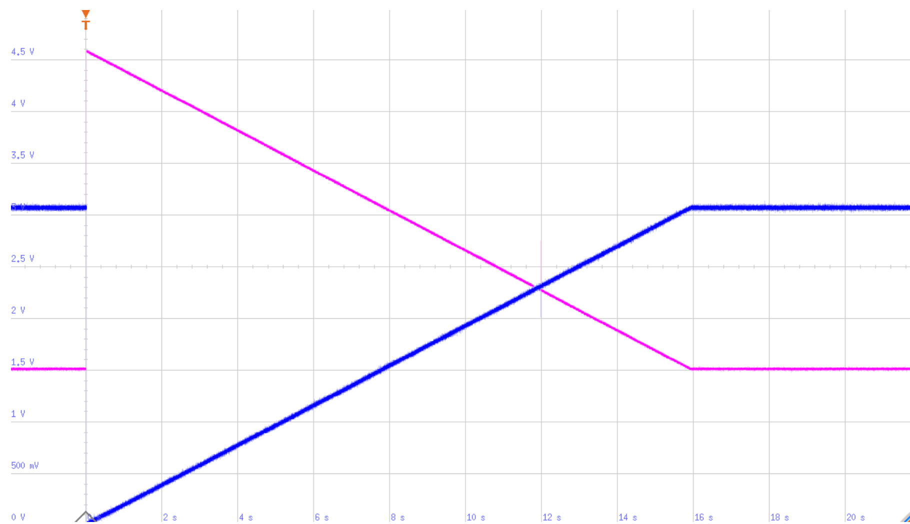


Figure 56: Measurement of ODAC0 and ODAC2 when writing system registers controlling the ODACs. The blue line is ODAC0 and the pink is ODAC2. The pink measurement is offset by ≈1.5V.

4.8.2 Configuration registers write and read

As mentioned in Section 3.8.2, the read function is for now limited to 1 byte at a time to avoid a bug in NM2. Verifying that the NM2 read function works correctly is done by writing to a register and then comparing the value returned with the value written. This is done extensibly to all configuration registers. The function bus_reg_wr_test() in appendix B iterates through all 235 system registers on the NM2 and inverts every bit of the registers marked with w/r. Then all registers are written one by one and then all registers are read back to match the memory map written to the NM2 with the one received. After running the function, the result is shown in Figure 57, which is run with a baud rate of 9600. The test shows that writing and reading all registers worked as there were no exception thrown in the python function when comparing the memory maps.

```
Bus register access Test started [full test].
** PASS ** All bus registers were written and read correctly.
Write x1 and read x2 of 232 registers took: 5.4531 seconds
```

Figure 57: Write and read test of all configuration registers with baud rate 9600.

4.8.3 Coefficient and RAM write and read

As mentioned in Section 2.2, there are three memory spaces on the NM2. The configuration registers are tested in the previous section, but the instruction RAM and the coefficient memory needs to be both written and read. The coefficient memory has a starting address of 1024 and 32 memory spaces per ADC, but only 19 used per ADC. This means there are $19 * 16 = 304$ registers to write and read to test that all memory spaces can be written correctly. This is done using WR_coeff() in appendix B. The function writes incrementing values to each register, that is on address 1024, 1025... 1042 for ADC0 and 1056, 1057... for ADC1. The result after

writing and reading all coefficient registers is shown in Figure 58. As the function ran without error, all coefficient registers were written and read correctly.

```
Write and read of coefficient memory started:
** PASS ** Coefficient memory correctly written and read.
Read and write coeffs: 16*19 = 304 registers took:  3.7043 seconds
```

Figure 58: Write and read test of all coefficient registers with baud rate 9600.

The same test is done to the instruction RAM. The RAM has addresses from 8192 to 12288 which is a total of 4096 registers. However, there is a bug present on the read of the RAM which makes all registers with odd addresses not readable. To avoid this, the test done in WR_ram() is only writing and reading from the even addresses in the RAM memory. The results is shown in Figure 59. The even addresses in the RAM was successfully written and read.

```
Write and read of even RAM registers started:
** PASS ** RAM write and read.
Write and read RAM: 4096/2 = 2048 registers took:  28.1127 seconds
```

Figure 59: Write and read test of all even RAM registers with baud rate 9600.

The results in this section show that the interfacing with NM2 works and that all three memory spaces can be accesses by the NM2 SPI interface. Total time for writing and reading all registers above is 37.6 s with baud rate 9600. If all RAM registers were included, this would in total take about $37.6\,\mathrm{s} + 28.1\,\mathrm{s} = 65.7\,\mathrm{s}$.

4.8.4 SPI reset and IO read

Both the SPI reset and IO read functions were not tested in conjunction with the NM2 ASIC due to the NM2RX IP not being implemented and therefore out of scope for this thesis. The functions were only verified using the oscilloscope. SPI reset(**011**) did send two bytes: **01110110** + **01110000**. SPI IO read(**010**) sent **01000000** and held CS low until another SPI IO read was sent.

4.9 Performance testing

In the Matrox Frame Grabber data sheet[Mat21], the range of the UART interface's baud rate is 9600 - 115200. Therefore after verifying that the system is stable with baud rate 9600, the CLOCKS_PER_BIT parameter in the UART TX and RX modules were changed to $\frac{100\,\mathrm{MHz}}{115\,200\,\mathrm{baud}} = 868$. This is stated as the maximum baud rate by Matrox. The same tests that were run in the previous section is now run with a baud rate of 115200. The results are shown in Figure 60 and passes. This verifies that the system is stable with a baud rate of 115200 and that the total time to write and read the registers is 3.7 s.

Even though 115200 is stated in the data sheet to be the maximum baud rate, baud rates higher than that works. To find out what the highest baud rate accepted by the frame grabber, a list of common baud rates are used and attempted to open the

```
Bus register access Test started [full test].
** PASS ** All bus registers were written and read correctly.
Write x1 and read x2 of 232 registers took: 0.5083 seconds

Write and read of coefficient memory started:
** PASS ** Coefficient memory correctly written and read.
Read and write coeffs: 16*19 = 304 registers took:  0.3363 seconds

Write and read of even RAM registers started:
** PASS ** RAM write and read.
Write and read RAM: 4096/2 = 2048 registers took:  2.5600 seconds
Complete read and write took:  3.7329 seconds
```

Figure 60: Full write and read of all registers with baud rate 115200.

serial port in windows. The list is copied from the PySerial documentation and used to check what the maximum accepted baud rate is. The python code is shown in Figure 61. After running the code, all baud rates up to 460800 can be configured, but none above.

```python
commonBauds = 50, 75, 110, 134, 150, 200, 300, 600, 1200, 1800, \
    2400, 4800, 9600, 19200, 38400, 57600, 115200, 230400, 460800, \
        500000, 576000, 921600, 1000000, 1152000, 1500000, 2000000, \
            2500000, 3000000, 3500000, 4000000

for baud in commonBauds:
    try:
        ser.baudrate = baud
        ser.open()
        ser.close()
        print("Baud ok: "+ str(baud))
    except:
        print("Baurate failed to configure: " + str(baud))
```

Figure 61: Python code for finding highest baud rate accepted by frame grabber.

The system is lastly tested using baud rate 460800, which is stated to exceed the limit of the frame grabber. After testing the full write and read of all registers, it still worked as expected. A comparison of the threes baud rates tested is given in Table 14. The last column is the estimated time it takes when all RAM registers are included and is calculated by doubling the RAM write and read time. Full test outputs are available in appendix C.

Baud rate	Write time	Read time	Write and read time	Estimated write and read
9600	10.7 s	24.6 s	28.1 s	65.7 s
115200	1.0 s	2.5 s	3.7 s	6.3 s
460800	0.3 s	0.9 s	1.3 s	2.0 s

Table 14: Speed comparison of write and read from all memory spaces.

4.9.1 Performance of previous firmware and software

No exact measurements of the previous firmware's performance is done in context of this thesis. However, it is reported at the IDEAS office that the previous firmware with the IDEAS testbench took about 10 minutes to write and read all memory spaces in full. This was improved by using python to write and read the Ethernet TCP. This reportedly took about 45 seconds for a full write and read of all registers.

5 Discussion

In the list of requirements(Section 1.1) of the thesis there are several requirements related to the integration of both the Camera Link IP and the NM2RX IP. As of the end of the semester, both of these IP's were not ready to be integrated in the firmware developed. This has led to a shifted end goal for the thesis. The focus shifted more into testing the limits of the system and comparing speeds to previous implementations of firmware targeting NM2.

A concern raised during the development of the firmware is how to implement both the UART and the SPI interfaces. The previous firmware implementations have all used an Ethernet cable to configure and communicate with the Zynq SoC and an SPI peripheral through the processor system. This implementation required embedded programming in C to control both the SPI and Ethernet. In addition to the serial ports being connected to the PL I/O pins. The focus of the master's project was using VHDL and therefore the decision was to implement the entire firmware in RTL in the PL of the Zynq. Doing this simplified the firmware implementation as there were no need to learn the embedded programming of the processor system. Even though the results of the system proved to be satisfactory, an engineer with experience with embedded systems would probably implement the SPI and UART faster and more reliable as there already are controllers for both interfaces that can be accessed by through the MIO pins. Then again, an engineer familiar with hardware design using VHDL or Verilog would maybe opt to stay away from embedded programming. As the choice was made to implement all in the PL part of the Zynq, the question of which FPGA used was raised. Not using the processor would mean that the whole firmware could be implemented on a simpler and thus cheaper FPGA family like the Spartan or Artix from AMD. This would lower the cost of development for future versions of NM2DB while also covering all the functionality needed.

A thought when trying to optimize the speed of the firmware was to increase the system clock frequency. The system clock frequency of 100 MHz is chosen somewhat arbitrarily as this was the frequency used in the NM2 RX IP. However, looking at the way the system is implemented, the limiting factors to efficiency are only the speed of the UART and the speed of the SPI. This is due to the system not having any complex calculations. All operations are synchronized on the system clock with no timing issues and all internal signals are transferred between registers in less than one clock period. Thus, increasing the speed of the system clock does not seem to have much impact of the performance. Increasing the SPI clock however, could lead to some improvement. NM2 has given a maximum SPI clock frequency of 20 MHz. A SPI function would require about 3.2 µs to write to and 11.2 µs to read a register. To write a 4 byte command through UART it takes 78 µs. Thus to read it would take $78 + 11.2 + 19 = 108.2$ µs in total. Doubling the SPI frequency would reduce this to $108 - 11.2/2 = 102.4$, a 5.5% decrease in time. Using the same calculations for a write operation, there is only a 2% decrease. The increase of SPI frequency would then be a minimal improvement. If there is a need to push the limits even further, with a total write and read time of 2s already, is hard to see.

Overall, the results of the firmware implementation is satisfactory in terms of speed and performance. Although no tests were done with the unfinished IPs, which led

to some requirements not being fulfilled, the implementation should be fairly easy. As the firmware and software outperforms earlier implementation, it is likely that some of the code will be used in the future by IDEAS.

Bibliography

[Tex14] Texas Instruments. 'tps7a47.pdf'. In: (2014).

[AIA18] AIA. 'Camera_Link_v2_1_Sept-20-2018.pdf'. In: (2018).

[Xil18] Xilinx. '7 Series FPGAs SelectIO Resources User Guide (UG471)'. In: (2018).

[Mat21] Matrox. 'Matrox Radient eV Installation and Hardware Reference'. In: (2021).

[Lie23] Chris Liechti. *PySerial documentation*. 2023. URL: https://pyserial.readthedocs.io/ (visited on 06/12/2023).

[Øfs23] Gjermund Øfsti. 'NIRCA MkII Development Board'. In: (2023).

[Tre23] Trenz Electronic. *TE0720 TRM*. 2023. URL: https://wiki.trenz-electronic.de/display/PD/TE0720+TRM (visited on 05/05/2023).

[Wik23] Wikipedia. *Camera Link*. 2023. URL: https://en.wikipedia.org/wiki/Camera_Link (visited on 26/11/2023).

Appendix

A Vivado simulations

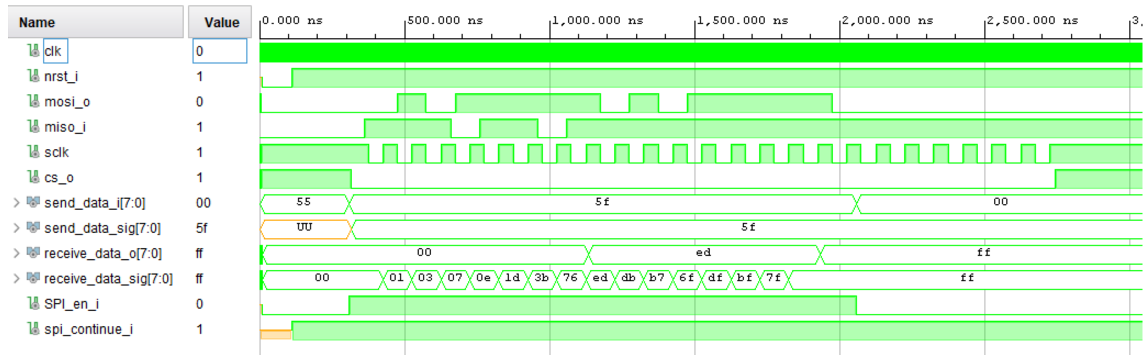


Figure 62: SPI master simulation result.

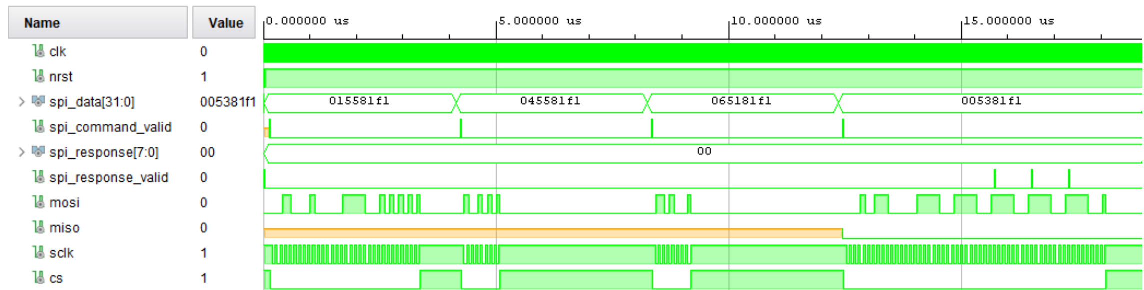


Figure 63: SPI handler simulation result.

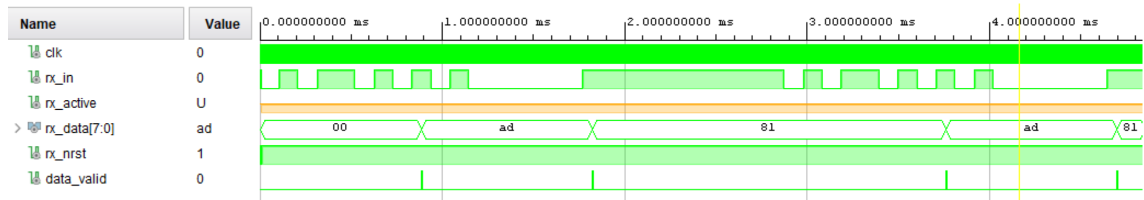


Figure 64: UART TX simulation result.

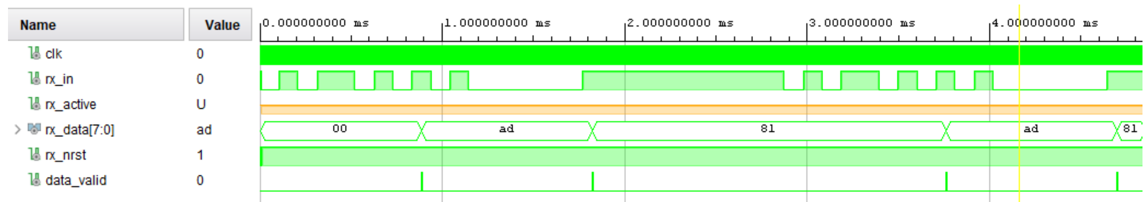


Figure 65: UART RX simulation result.

B nm2_ser_config.py

```
from os import path
import sys
import time
from memmap_nm2 import MMRST, ADC, TX, ACQ
import serial
import serial.tools.list_ports
import fw_constants as const
import math
import struct
import numpy as np
import operator
MM_CONF = MMRST.copy()

ser = serial.Serial()

def open_serial():
    ser.timeout = 1 #If no bytes are returned, this hangs the
    ↪ program
    #BAUDRATE CHOICE:
    #ser.baudrate = 9600 #10417 clocks per bit
    #ser.baudrate = 115200 #868 clocks per bit
    ser.baudrate = 460800 #217 clocks per bit

    ser.port = 'COM5'
    ser.open()
    time.sleep(0.1)
    ser.read(1000) #Empty buffer
    print("Attempting to open port: "+str(ser.port) + " with baud
    ↪ rate: " + str(ser.baudrate))
    print("Serial port is open?: " + str(ser.is_open))

def LDO_enable(value):
    packet = bytearray()
    packet.append(const.NM2_LDO_CONFIG)
    packet.append(value)
    #packet.append(0x0f) # Enables all LDOs
    ser.write(packet)

def LDO_read():
    packet = bytearray()
    packet.append(const.NM2_LDO_READ)
    ser.write(packet)
    value = ser.read(1)
    return int(value.hex(),16)

def LDO_disable():
    packet = bytearray()
```

```
    packet.append(const.NM2_LDO_CONFIG)
    packet.append(0x00) # Disables all LDOs
    ser.write(packet)

def AVDDH0_write(value):
    packet = bytearray()
    packet.append(const.NM2_AVDDH0_WRITE)
    packet.append(value)
    ser.write(packet)

def AVDDH0_read():
    packet = bytearray()
    packet.append(const.NM2_AVDDH0_READ)
    ser.write(packet)
    value = ser.read(1)
    return int(value.hex(),16)

def AVDDH1_write(value):
    packet = bytearray()
    packet.append(const.NM2_AVDDH1_WRITE)
    packet.append(value)
    ser.write(packet)

def AVDDH1_read():
    packet = bytearray()
    packet.append(const.NM2_AVDDH1_READ)
    ser.write(packet)
    value = ser.read(1)
    return int(value.hex(),16)

def NM2_clock_select(value): #0: disable, 1: 15MHz, 2: 12MHz, 3:
↪   10MHz, 4: 6MHz
    packet = bytearray()
    packet.append(const.NM2_CLOCK_SELECT)
    packet.append(value)
    ser.write(packet)

def NM2_write(addr, value):
    packet = bytearray()
    packet.append(const.NM2_SPI_WRITE) #Write command
    packet.append(value)
    packet.append(int(addr/256))
    packet.append(addr%256)
    ser.write(packet)

def NM2_read(addr, fieldpos):
    packet = bytearray()
    packet.append(const.NM2_SPI_READ) #Read command
    packet.append(0x01) #Only read one byte
    packet.append(int(addr/256))
```

```
    packet.append(addr%256)
    ser.write(packet)
    value = ser.read(1)
    return int(value.hex(),16)

def NM2_init_setup():

    # Write reg0
    packet = bytearray()
    packet.append(const.NM2_SPI_WRITE_REG0)
    initValueReg0 = 0x05
    packet.append(initValueReg0) # initial value
    ser.write(packet)
    # Read reg0
    packet = bytearray()
    packet.append(const.NM2_SPI_READ_REG0)
    ser.write(packet)
    value = ser.read(1)
    value = struct.unpack('>H', b'\x00' + value)[0]
    if(value != initValueReg0):
        raise Exception("Reg0 not written correctly")
    # Write reg1
    packet = bytearray()
    packet.append(const.NM2_SPI_WRITE_REG1)
    initValueReg1 = 0x12
    packet.append(initValueReg1)
    ser.write(packet)
    # Read reg1
    packet = bytearray()
    packet.append(const.NM2_SPI_READ_REG1)
    ser.write(packet)
    value = ser.read(1)
    value = struct.unpack('>H', b'\x00' + value)[0]
    if(value != initValueReg1):
        raise Exception("Reg1 not written correctly")


def ODAC_enable():
    MM_CONF.fields['odac_enable'].write_value(255, hook = NM2_write)

def ODAC0_write(value):
    MM_CONF.fields['odac0_dac_lo'].write_value(value%256, hook =
    ↪   NM2_write)
    MM_CONF.fields['odac0_dac_hi'].write_value(int(value/256), hook =
    ↪   NM2_write)

def ODAC2_write(value):
    MM_CONF.fields['odac2_dac_lo'].write_value(value%256, hook =
    ↪   NM2_write)
```

```
    MM_CONF.fields['odac2_dac_hi'].write_value(int(value/256), hook =
     ↪  NM2_write)

##TEST FUNCTIONS FOR SPEED COMPARISON##
def write_coeff():
    ##Testing write of all coeff registers.
    ticWrite = time.perf_counter()
    for i in range(15):
        for j in range(18):
            addr = 1024 + (i)*32 + j
            NM2_write(addr,j)
    tocWrite = time.perf_counter()
    print(f"Only write coeffs: 16*19 = 304 registers took:  {tocWrite
     ↪  - ticWrite:0.4f} seconds")

def read_coeff():
    ##Testing write of all coeff registers.
    ticRead = time.perf_counter()
    for i in range(15):
        for j in range(18):
            addr = 1024 + (i)*32 + j
            NM2_read(addr,1)
    tocRead = time.perf_counter()
    print(f"Only read coeffs: 16*19 = 304 registers took:  {tocRead -
     ↪  ticRead:0.4f} seconds")

def WR_coeff():
    ##Testing write and read of all coeff registers.
    print("\nWrite and read of coefficient memory started: ")
    ticWR = time.perf_counter()
    for i in range(15):
        for j in range(18):
            addr = 1024 + (i)*32 + j
            NM2_write(addr,j)
            value = NM2_read(addr,1)
            if(value != j):
                print(value)
                print(j)
                raise Exception("Read value: " + str(value) + " does
                 ↪  not match the written: "
                                + str(j))
    tocWR = time.perf_counter()
    print("** PASS ** Coefficient memory correctly written and
     ↪  read.")
    print(f"Read and write coeffs: 16*19 = 304 registers took:
     ↪  {tocWR - ticWR:0.4f} seconds")

def write_ram():
    ramAddresses = np.linspace(8192, 12288, 2049)
    ramAddresses = [int(x) for x in ramAddresses]
```

```
    i=0
    ticWram = time.perf_counter()
    for address in ramAddresses:
        i+=1
        if i > 200:
            i = 0
        NM2_write(address,i)
    tocWram = time.perf_counter()

    print(f"Only write RAM: 4096/2 = 2048 registers took:  {tocWram -
    ↪  ticWram:0.4f} seconds")

def read_ram():
    ramAddresses = np.linspace(8192, 12288, 2049)
    ramAddresses = [int(x) for x in ramAddresses]
    ticRram = time.perf_counter()
    for address in ramAddresses:
        NM2_read(address,1)
    tocRram = time.perf_counter()
    print(f"Only read RAM: 4096/2 = 2048 registers took:  {tocRram -
    ↪  ticRram:0.4f} seconds")

def WR_ram(start):
    print("\nWrite and read of even RAM registers started: ")
    i=start
    ramAddresses = np.linspace(8192, 12288, 2049)
    ramAddresses = [int(x) for x in ramAddresses]
    ticWRram = time.perf_counter()
    for address in ramAddresses:
        i+=1
        if i > 200:
            i = 0
        NM2_write(address,i)
        value = NM2_read(address,1)
        if(value != i):
            raise Exception("ADDR: "+ str(address) + "Read value: " +
            ↪  str(value) +
                                " does not match the written: " + str(i))
    tocWRram = time.perf_counter()
    print("** PASS ** RAM write and read.")
    print(f"Write and read RAM: 4096/2 = 2048 registers took:
    ↪  {tocWRram - ticWRram:0.4f} seconds")

def bus_reg_wr_test():
    ''' Write / Read / Read test of bus registers '''
    # Init ASIC
    MM_CONF_WRITE = MMRST.copy()    #MemoryMap containing the write
    ↪  data
    MM_CONF_READ1 = MMRST.copy()    #MemoryMap containing the read
    ↪  data from first read
```

```
MM_CONF_READ2 = MMRST.copy()     #MemoryMap containing the read
 ↪  data from second read
sorted_tuples                             =
 ↪  (sorted(MM_CONF_WRITE.fields.values(),

                                                               ↪  key=operator.attrgetter('addr'))
## Populate MM with test data
for f in sorted_tuples:
    if f.fields[0]['addr'] == 0:
        continue # IRQ_ENABLE address used for some Testbenchs
         ↪  stuff
    elif f.fields[0]['addr'] == 0xFC00:
        continue # SPI_RESET not a register
    elif f.fields[0]['addr'] == 0x00EA:
        continue # Holds the ref_clk_disable bit
    elif f.fields[0]['romask'] == 1:
        continue # Skip ReadOnly
    elif f.fields[0]['rrmask'] == 1:
        continue # Skip ReadReset
    elif f.fields[0]['pumask'] == 1:
        continue # Skip PulseReg
    elif f.fields[0]['wrxmask'] == 1:
        continue # Skip write/read/ext since write only
         ↪  manifests after sequencer sync reset
    elif f.fields[0]['logical'] == 1:
        continue # Skip logical bitfields
    elif 'no_reg' in f.name:
        continue # Skip 'no_reg' empty bitfields (not actual
         ↪  registers)
    else:
        # Populate all w/r registers
        if f.fields[0]['addr'] < 235:        # Limit to bus
         ↪  registers only
            bitwise_inverted_reset_data = \

                 ↪  abs(int('{:08b}'.format(-255-(~MMRST.fields[f.name].get_val
            MM_CONF_WRITE.fields[f.name].set_value(
             ↪  bitwise_inverted_reset_data )
print("\nBus register access Test started [full test].")
# Write/read test of bus registers
ticBusReg = time.perf_counter()
for addr in range(1,236):
    if addr != 234:
        MM_CONF_WRITE.registers[ addr ].write_reg( hook =
         ↪  NM2_write )
        read_data1 = MM_CONF_READ1.registers[ addr ].read_reg(
         ↪  hook = NM2_read )
        read_data2 = MM_CONF_READ2.registers[ addr ].read_reg(
         ↪  hook = NM2_read )
tocBusReg = time.perf_counter()
```

```
    compare_map1_ok = MM_CONF_WRITE.compare_map(MM_CONF_READ1)
    compare_map2_ok = MM_CONF_WRITE.compare_map(MM_CONF_READ2)
    if compare_map1_ok == True and compare_map2_ok == True:
        print("** PASS ** All bus registers were written and read
         ↪ correctly.")
        print(f"Write x1 and read x2 of 232 registers took:
         ↪ {tocBusReg - ticBusReg:0.4f} seconds")
    #
     ↪ ----------------------------------------------------------------------

def bus_reg_w_test():
    ''' Write / Read / Read test of bus registers '''
    # Init ASIC
    MM_CONF_WRITE = MMRST.copy()     #MemoryMap containing the write
     ↪ data
    sorted_tuples                    =
     ↪ (sorted(MM_CONF_WRITE.fields.values(),

                                                         ↪ key=operator.attrgetter('addr')))
    ## Populate MM with test data
    for f in sorted_tuples:
        if f.fields[0]['addr'] == 0:
            continue # IRQ_ENABLE address used for some Testbenchs
             ↪ stuff
        elif f.fields[0]['addr'] == 0xFC00:
            continue # SPI_RESET not a register
        elif f.fields[0]['addr'] == 0x00EA:
            continue # Holds the ref_clk_disable bit
        elif f.fields[0]['romask'] == 1:
            continue # Skip ReadOnly
        elif f.fields[0]['rrmask'] == 1:
            continue # Skip ReadReset
        elif f.fields[0]['pumask'] == 1:
            continue # Skip PulseReg
        elif f.fields[0]['wrxmask'] == 1:
            continue # Skip write/read/ext since write only
             ↪ manifests after sequencer sync reset
        elif f.fields[0]['logical'] == 1:
            continue # Skip logical bitfields
        elif 'no_reg' in f.name:
            continue # Skip 'no_reg' empty bitfields (not actual
             ↪ registers)
        else:
            # Populate all w/r registers
            if f.fields[0]['addr'] < 235:         # Limit to bus
             ↪ registers only
                bitwise_inverted_reset_data = \

                     ↪ abs(int('{:08b}'.format(-255-(~MMRST.fields[f.name].get_val
```

```
                MM_CONF_WRITE.fields[f.name].set_value(
                 ↪  bitwise_inverted_reset_data )
    print("\nBus register access Test started [only write].")
    # Write/read test of bus registers
    ticBusReg = time.perf_counter()
    for addr in range(1,236):
        if addr != 234:
            MM_CONF_WRITE.registers[ addr ].write_reg( hook =
             ↪  NM2_write )
    tocBusReg = time.perf_counter()

    print(f"Write x1 232 registers took: {tocBusReg - ticBusReg:0.4f}
     ↪  seconds")
    #
     ↪  -----------------------------------------------------------------------

def bus_reg_r_test():
    ''' Write / Read / Read test of bus registers '''
    # Init ASIC
    MM_CONF_READ1 = MMRST.copy()    #MemoryMap containing the read
     ↪  data from first read
    MM_CONF_READ2 = MMRST.copy()    #MemoryMap containing the read
     ↪  data from second read
    print("\nBus register access Test started [only read once].")
    # Write/read test of bus registers
    ticBusReg = time.perf_counter()
    for addr in range(1,236):
        if addr != 234:
            read_data1 = MM_CONF_READ1.registers[ addr ].read_reg(
             ↪  hook = NM2_read )
    tocBusReg = time.perf_counter()
    print(f"Read of 232 registers took: {tocBusReg - ticBusReg:0.4f}
     ↪  seconds")
    #
     ↪  -----------------------------------------------------------------------
```

C Test reports

C.1 TestFullConfig9600.txt

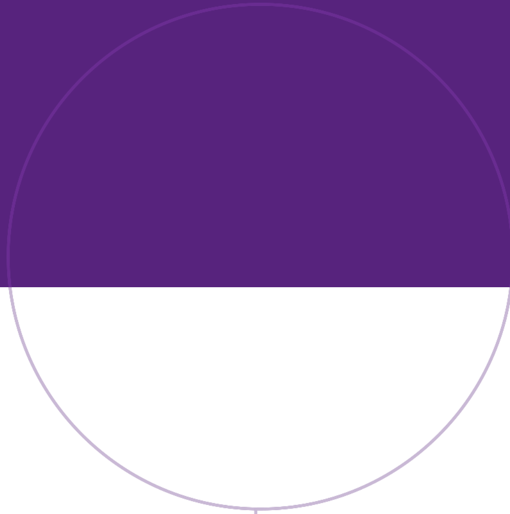
```
SETUP:

Attempting to open port: COM5 with baud rate: 9600
Serial port is open?: True
Initial setup OK!

TESTS:

Bus register access Test started [only write].
Write x1 232 registers took: 0.9702 seconds
Only write RAM: 4096/2 = 2048 registers took:  8.5370 seconds
Only write coeffs: 16*19 = 304 registers took:  1.1247 seconds
Complete write took:  10.7337 seconds

Bus register access Test started [only read once].
Read of 232 registers took: 2.2398 seconds
Only read RAM: 4096/2 = 2048 registers took:  19.5592 seconds
Only read coeffs: 16*19 = 304 registers took:  2.5769 seconds
Complete read took:  24.5688 seconds

Bus register access Test started [full test].
** PASS ** All bus registers were written and read correctly.
Write x1 and read x2 of 232 registers took: 5.4531 seconds

Write and read of coefficient memory started:
** PASS ** Coefficient memory correctly written and read.
Read and write coeffs: 16*19 = 304 registers took:  3.7043 seconds

Write and read of even RAM registers started:
** PASS ** RAM write and read.
Write and read RAM: 4096/2 = 2048 registers took:  28.1127 seconds
Complete read and write took:  37.6013 seconds
```

C.2 TestFullConfig115200.txt

```
SETUP:

Attempting to open port: COM5 with baud rate: 115200
Serial port is open?: True
Initial setup OK!

TESTS:

Bus register access Test started [only write].
Write x1 232 registers took: 0.0815 seconds
Only write RAM: 4096/2 = 2048 registers took:  0.7111 seconds
Only write coeffs: 16*19 = 304 registers took:  0.0936 seconds
Complete write took:  0.9908 seconds

Bus register access Test started [only read once].
Read of 232 registers took: 0.2176 seconds
Only read RAM: 4096/2 = 2048 registers took:  1.8637 seconds
Only read coeffs: 16*19 = 304 registers took:  0.2444 seconds
Complete read took:  2.5211 seconds

Bus register access Test started [full test].
** PASS ** All bus registers were written and read correctly.
Write x1 and read x2 of 232 registers took: 0.5083 seconds

Write and read of coefficient memory started:
** PASS ** Coefficient memory correctly written and read.
Read and write coeffs: 16*19 = 304 registers took:  0.3363 seconds

Write and read of even RAM registers started:
** PASS ** RAM write and read.
Write and read RAM: 4096/2 = 2048 registers took:  2.5600 seconds
Complete read and write took:  3.7329 seconds
```

C.3 TestFullConfig460800.txt

```
SETUP:
Attempting to open port: COM5 with baud rate: 460800
Serial port is open?: True
Initial setup OK!

TESTS:

Bus register access Test started [only write].
Write x1 232 registers took: 0.0212 seconds
Only write RAM: 4096/2 = 2048 registers took:  0.1829 seconds
Only write coeffs: 16*19 = 304 registers took:  0.0241 seconds
Complete write took:  0.3310 seconds

Bus register access Test started [only read once].
Read of 232 registers took: 0.0652 seconds
Only read RAM: 4096/2 = 2048 registers took:  0.5390 seconds
Only read coeffs: 16*19 = 304 registers took:  0.0723 seconds
Complete read took:  0.8724 seconds

Bus register access Test started [full test].
** PASS ** All bus registers were written and read correctly.
Write x1 and read x2 of 232 registers took: 0.1506 seconds

Write and read of coefficient memory started:
** PASS ** Coefficient memory correctly written and read.
Read and write coeffs: 16*19 = 304 registers took:  0.0966 seconds

Write and read of even RAM registers started:
** PASS ** RAM write and read.
Write and read RAM: 4096/2 = 2048 registers took:  0.7250 seconds
Complete read and write took:  1.2900 seconds
```



NTNU
Norwegian University of
Science and Technology

ideas
Integrated Detector Electronics AS