JOHANNES GUTENBERG
UNIVERSITÄT MAINZ

Module Control of the jFEX for the ATLAS Calorimeter Trigger Upgrade

by

Rouven Spreckels

n3vu0r@qu1x.org

Diploma Thesis in Physics

3rd June 2016

Supervisor

Prof. Dr. Stefan Tapprogge

Abstract

The ATLAS experiment at the Large Hadron Collider (LHC) is a general-purpose particle detector searching for new fundamental physics discoveries. The LHC and the ATLAS detector will be upgraded to reach higher luminosities and finer granularities, respectively. To maintain trigger efficiency, new subsystems will be installed as part of the upgrade, such as the Jet Feature Extractor (jFEX). It will identify in real-time jets and τ particles and calculate energy sums with the data received from electromagnetic and hadronic calorimeters by running its algorithms on multiple processor FPGAs. The implementations and configurations of these algorithms are provided by a single control FPGA accessed through a central control interface. For reasons of flexibility, this control FPGA is placed on a mezzanine card based on a hybrid system on a chip (SoC), combining an FPGA and a CPU inside a single chip with many interconnects in between. This thesis presents the design of this mezzanine card and the software developed to demonstrate use cases of the hybrid SoC approach.

Statutory Declaration

I hereby confirm that I have written the present thesis independently and without illicit assistance from third parties and using solely the aids mentioned.

Mainz, 3rd June 2016

Rouven Spreckels

Contents

Chapter 1

Overview

The ATLAS experiment [1] at the Large Hadron Collider (LHC) [2] is a general-purpose particle detector searching for new fundamental physics discoveries while further investigating the properties of its recent discoveries [3], whether predicted by the Standard Model or beyond of it. To study elementary particles and their fundamental interactions, they are accelerated to high kinetic energies in order to let them collide and to analyze their products. The ATLAS detector [1] generates data for each detected event resulting in such high data rates, unmanageable to be stored for long-term analyzes. Thus, a trigger system [4] is used to select in real-time the rare interesting events giving hints to new physics, reducing the effective data rates down to a manageable level. In order to make trigger decisions in real-time, custom-made electronics are used to meet the necessary discriminatory power. The LHC and the ATLAS detector were and will further be upgraded to reach higher luminosities and finer granularities, respectively [5]. Thus, the trigger system was and will further be upgraded as well to manage the increased data rates generated by the detector. The trigger system is divided into calorimeter and muon triggers. The Jet Feature Extractor (jFEX) [6] currently being designed will augment the modules of the present calorimeter trigger as part of the next upgrade to meet the need for more discriminatory power. It will identify in real-time jets and τ particles and calculate energy sums with the data received from the electromagnetic and hadronic calorimeters of the detector by running its algorithms on multiple FPGAs, here referred to as processor FPGAs. The implementations and configurations of these algorithms are provided by a single control FPGA accessed through a central control interface. For reasons of flexibility, this control FPGA is placed on a daughter module, a so-called mezzanine card. This mezzanine card further controls several subcomponents of the jFEX. It populates a hybrid system on a chip (SoC),

combining an FPGA and a CPU inside a single chip with many interconnects in between. The following chapter 2 introduces the jFEX while describing its integration into the ATLAS experiment at the LHC. I have collaborated on the hardware design of the mezzanine card which is presented in chapter 3, while chapter 4 covers the software I have developed to demonstrate use cases of the hybrid SoC approach regarding the module control of the jFEX. Finally, I have tested the mezzanine card in combination with the developed software as described in chapter 5, followed by a conclusion and outlook in chapter 6.

Chapter 2

Introduction

The Standard Model of particle physics has successfully explained and predicted many experimental results like the discovery of the Higgs boson in 2012 [3]. But for beyond of what this theory is capable to explain like dark matter particles, ongoing research is necessary to further develop its description of elementary particles and their fundamental interactions. To study their laws of nature, particles are accelerated to high kinetic energies in order to let them collide and to analyze their products. Many interesting particles are only produced at high collision energies. The challenge is to select the rare interesting events giving hints to new physics.

2.1 Large Hadron Collider

The Large Hadron Collider (LHC) [2] is located at the European Organization for Nuclear Research (CERN)[1] near Geneva, Switzerland. As of writing time, it is the world's largest and most powerful particle collider, lying 100 meters beneath ground in a tunnel of 27 kilometers in circumference. It accelerates hadrons, specifically protons or lead ions, in two adjacent parallel beam pipes of opposite directions. These pipes intersect at four points allowing the beams to collide within the de-

[1] The original "Conseil Européen pour la Recherche Nucléaire" (CERN) was dissolved [7].

tectors of the four main experiments ATLAS[2], CMS[3], ALICE[4], and LHCb[5] each located at an intersection point, as shown in figure 2.1. First two are general-purpose particle detectors while last two are more specialized ones.

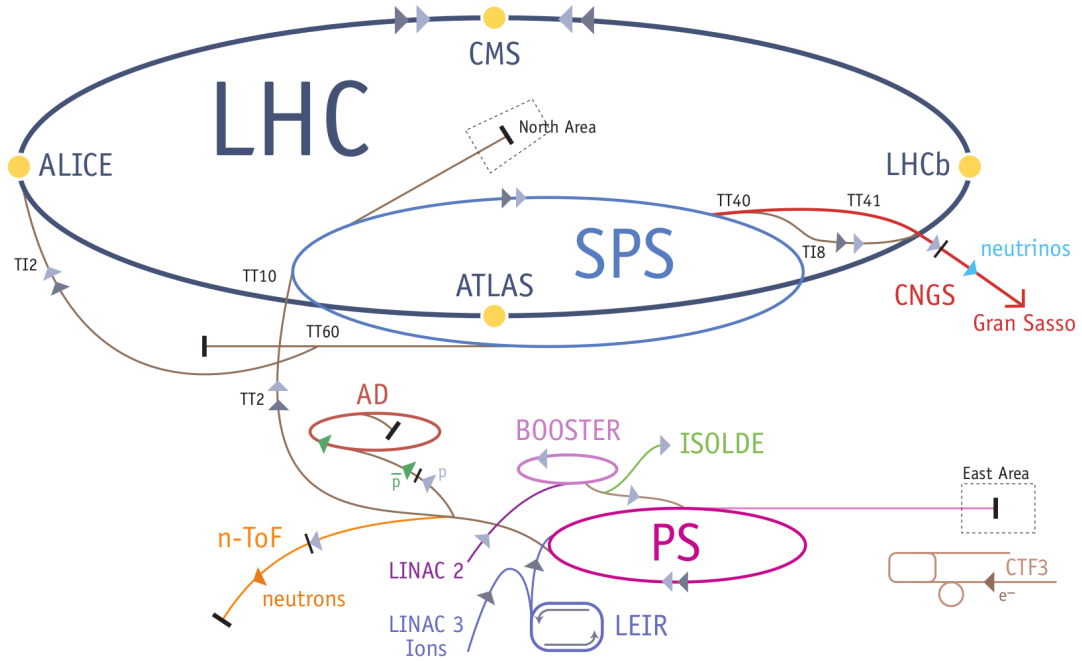


Figure 2.1: LHC Overview [2]

The beam parameters of the operational runs of proton-proton collisions are shown in table 2.1, which are the center-of-mass energy $\sqrt{s}$, the instantaneous luminosity[6] L, and the average number of interactions per bunch-crossing $\langle\mu\rangle$ along with its peak value μ_{max}. A long shutdown between the runs is used to install upgrades in order to reach the increased parameters.

[2] A Toroidal LHC ApparatuS (ATLAS) is a general-purpose detector experiment [2].
[3] The Compact Muon Solenoid (CMS) is a general-purpose detector experiment [2].
[4] A Large Ion Collider Experiment (ALICE) is a specialized detector for heavy ions [2].
[5] The LHC beauty (LHCb) experiment is a specialized detector for B mesons [2].
[6] The luminosity $L = \frac{1}{\sigma}\frac{dN}{dt}$ is the rate of events produced per second $\frac{dN}{dt}$ and cross-section σ.

Run	Years	$\sqrt{s}$ (TeV)	L ($10^{34}\,\mathrm{cm}^{-2}\mathrm{s}^{-1}$)	$\langle\mu\rangle$	μ_{max}
1	2010-2012	≤ 8	0.77	21	36
Phase-0 Upgrade during Long Shutdown 1 (LS1)					
2	2015-2017	13	~ 1.6	40	~ 60
Phase-1 Upgrade during Long Shutdown 2 (LS2)					
3	2020-2022	13	~ 2.5	60	~ 80
Phase-2 Upgrade during Long Shutdown 3 (LS3)					
4	2025-2027	14	~ 5.0	140	~ 200

Table 2.1: LHC Evolution [5]

2.2 ATLAS Experiment

The goals of the ATLAS experiment are to search for new discoveries and further investigate its recent discoveries like the properties of the Higgs boson [1]. Ongoing measurements will allow in-depth analyzes of the Standard Model and beyond of it.

2.2.1 Detector

The ATLAS detector [1] shown in figure 2.2 is composed of concentric cylinders surrounding an intersection point of the LHC where the proton beams collide. Its main components are the inner detector, the inner electromagnetic calorimeter (ECAL), the outer hadronic calorimeter (HCAL), and the outermost muon spectrometer. Each is specialized to detect certain types and properties of particles, complementing to a general-purpose detector in a whole by covering various particles with a broad range of energies.

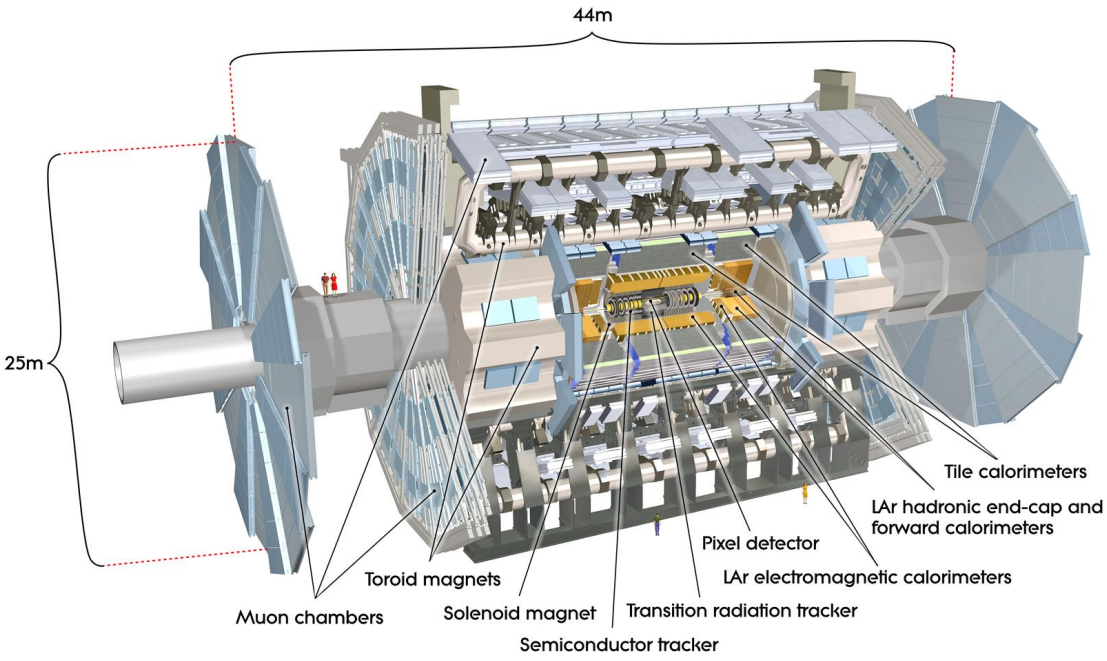


Figure 2.2: ATLAS Detector [1]

The inner detector is the closest to the proton beam with a gap of only a few centimeters. It tracks charged particles in order to reconstruct their trajectories. Due to a magnetic field within the inner detector, these trajectories are curved. Their direction and degree of curvature reveals the sign of the charge and the momentum of a particle, respectively. The two calorimeters measure energy absorptions of particle showers caused by interactions with their material. The initial energy of particles is inferred by sampling their showers. The inner calorimeter covers electromagnetic interactions and the outer calorimeter covers strong interactions. The muon spectrometer functions similarly to the inner detector by curving the trajectories of muons with a magnetic field in order to measure their momentum.

2.2.2 Trigger and Data Acquisition

According to the Technical Design Report (TDR) for the Phase-1 upgrade of the ATLAS Trigger and Data Acquisition (TDAQ) system [4], the LHC collides two bunches of protons every $25\,\mathrm{ns}$ resulting in 40 million bunch-crossings in a second, that is a bunch-crossing rate of $40\,\mathrm{MHz}$. For each bunch-crossing the data generated by the several sub-detectors sum up to about $2.4\,\mathrm{MB}$ after zero suppression. This results in an unmanageable data rate of roughly $40\,\mathrm{MHz} \cdot 2.4\,\mathrm{MB} = 96\,\mathrm{TB/s}$. Since not all

events are interesting, a pipelined trigger system of multiple stages is used to select in real-time the potentially interesting events and their regions of interest (RoIs) within the detector. This reduces the effective event rate and its corresponding data rate step by step down to a manageable level of roughly $2.4\,\mathrm{GB/s}$ to be stored for long-term analyzes, as shown in table 2.2.

Stage	Trigger Rate	Data Rate
Detector Read-Out	40 MHz	96 TB/s
Level-1 Accepts	100 kHz	240 GB/s
High-Level Requests	40 kHz	60 GB/s
Event Building	12 kHz	29 GB/s
Event Filtering	1 kHz	2.4 GB/s

Table 2.2: Trigger & Data Rates during Run 2 [4]

The trigger system is divided into two levels of event selection, the Level-1 (L1) and the High-Level Trigger (HLT). The L1 is implemented using custom-made electronics based on ASICs[7] and FPGAs[8] while the HLT is based on farms of commercially available computers and networking hardware. Each stage of table 2.2 reduces the effective event rate giving the algorithms of the next stage respectively more time to analyze the data with more complex algorithms and more precise selection criteria. The L1 trigger, divided into calorimeter trigger (L1Calo) and muon trigger (L1Muon), has less than $2.5\,\mu\mathrm{s}$ per bunch-crossing to make its decisions and to transfer its results to the HLT while it is concurrently fed by the data stream of the current bunch-crossing due to being a pipelined system.

2.2.3 Calorimeter Trigger

The Level-1 Calorimeter Trigger (L1Calo) currently being used during Run 2, that is after the Phase-0 upgrade [4], identifies e/γ and τ particles[9] with the data received from the ECAL and HCAL. Furthermore, it identifies jets and events of large missing transverse energy E_T^{miss} (XE) along with its significance XS, and large total transverse energy $\sum E_T$ (TE). The architecture of the L1Calo system is shown in figure 2.3.

[7] An application-specific integrated circuit (ASIC) is an IC manufactured for a custom use.
[8] A field-programmable gate array (FPGA) is an IC customizable after manufacturing.
[9] Not distinguishing between e^-, e^+, and γ while only identifying hadronically decaying τ.

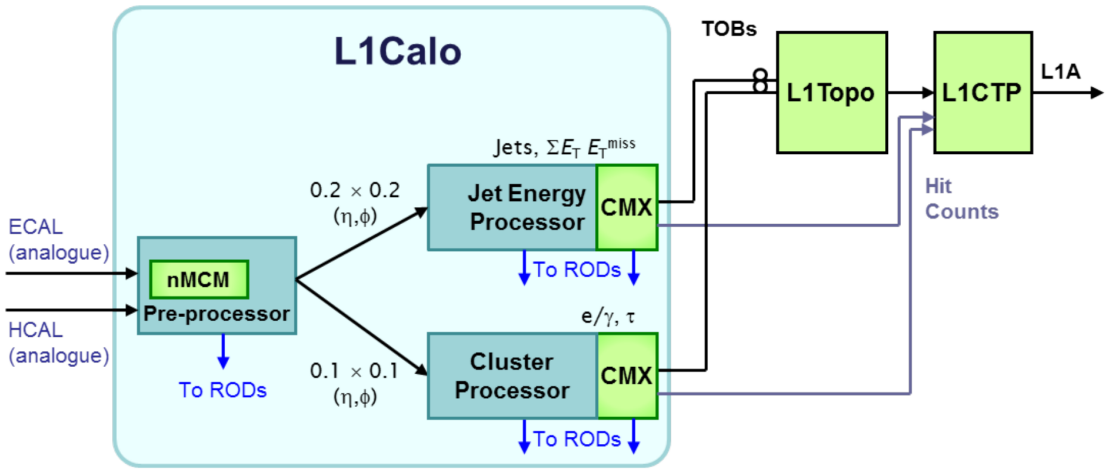


Figure 2.3: L1Calo System Architecture during Run 2 [4]

The L1Calo system receives analogue signals from the ECAL and HCAL. The Pre-Processor Modules (PPMs) digitize them with a sampling rate of $80\,\mathrm{MHz}$ and send them to the Cluster Processor (CP) subsystem and to the Jet Energy Processor (JEP) subsystem. Each subsystem consists of multiple modules, 56 Cluster Processor Modules (CPMs) and 32 Jet Energy Processor Modules (JEMs), respectively. They are installed in a crate and connected through its backplane. Each CPM identifies and counts energy deposits indicative of isolated e/γ and τ particles within a given area of the calorimeter data while each JEM identifies jets and calculates partial energy sums of missing transverse energy E_T^{miss} and total transverse energy $\sum E_T$. Both subsystems build Trigger Objects (TOBs) comprising location, energy, object type, XE, XS, and TE. They are sent at $160\,\mathrm{MHz}$ over the backplane of the crate to 12 Common Merger Extended Modules (CMXs), 8 for the CP subsystem and 4 for the JEP subsystem. The CMXs merge the TOBs of the two subsystems and route them optically to the L1 Topological Processor (L1Topo). Additionally, they send object counts to the L1 Central Trigger Processor (L1CTP). The L1Topo combines these TOBs with the ones of the L1Muon in order to make trigger decisions based on the full event topology. Its resulting decision bits are sent to the L1CTP. On a L1 Accept (L1A), that is the L1CTP considered events to be of interest, all modules of the subsystems are read-out via Read-Out Drivers (RODs). They collect all read-out data plus RoIs, manage data buffering and flow control, and build event packets to be sent to the HLT and Data Acquisition (DAQ) system.

2.2.4 Calorimeter Trigger Upgrade

The L1Calo system will be upgraded in the LS2 between Run 2 and 3 as part of the Phase-1 upgrade [4, 6]. According to table 2.1, the LHC will further increase its luminosity resulting in an increased number of interactions per bunch-crossing for Run 3. This will require more discriminatory power to maintain trigger efficiency. Thus, a new set of Feature Extractor (FEX) subsystems will be installed, processing calorimeter data of finer granularity with larger-area algorithms. It allows to select events more precisely, reject more background, and cover larger jets than with the JEP subsystem. Due to the increased number of interactions, improved pile-up corrections will be implemented as well. The architecture of the L1Calo system to be used during Run 3, that is after the Phase-1 upgrade, is shown in figure 2.4.

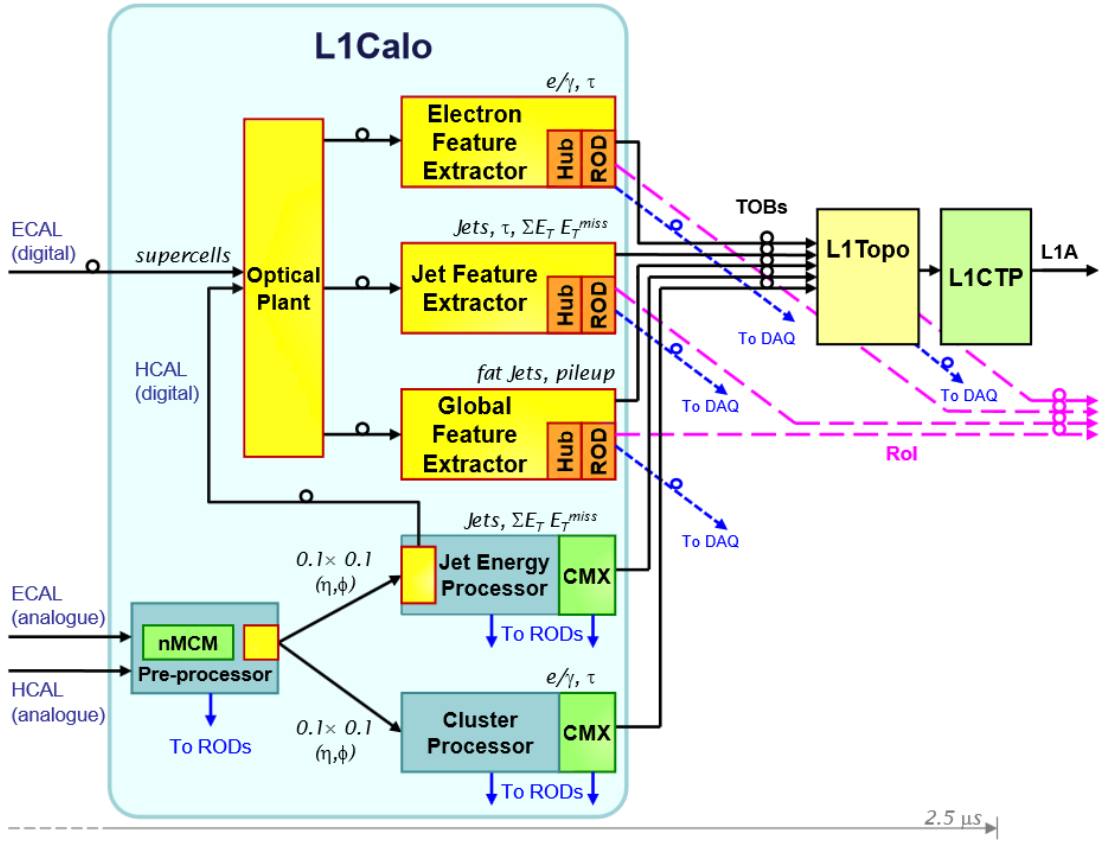


Figure 2.4: L1Calo System Architecture during Run 3 [6, modified]

The FEX subsystems are the electromagnetic Feature Extractor (eFEX) of 24 modules, the jet Feature Extractor (jFEX) of 7 modules, and the global Feature Extractor (gFEX) of a single module. The eFEX and jFEX will replace the CP and JEP, respectively while the gFEX will identify features requiring the whole calorimeter data.

The FEX subsystems will operate in parallel with the CP and JEP subsystems until their outputs will have been validated. To meet the requirements of higher throughput rates due to data of finer granularity, the ECAL will provide signals over optical fibers, digitized and duplicated for each FEX module by a new on-detector Digital Processing System (DPS). In contrast, the HCAL signals will be digitized by the PPMs and duplicated by a new daughter module of the JEMs. The optical fibers will be routed to an optical plant in order to map them in such a way that each eFEX and jFEX module is fed with data partially overlapping with the data of neighboring modules. This inter-module redundancy is required by their algorithms, processing the data in moving intervals of two dimensions, so-called sliding windows. In contrast, the single module of the gFEX subsystem is fed with the whole calorimeter data.

2.3 Jet Feature Extractor

As part of the ATLAS calorimeter trigger upgrade of section 2.2.4, the jFEX subsystem [6] will identify in real-time large energy deposits and calculate missing transverse energies E_T^{miss} and total transverse energies $\sum E_T$ with the data received from the electromagnetic and hadronic calorimeters. These energy deposits are indicative of jets and τ particles. The jFEX subsystem comprises 7 jFEX modules. Each will run multiple versions of the sliding window algorithm in parallel on its four processor FPGAs in order to identify the energy deposits from the calorimeter data, as described in its specification draft [6]. They will be installed in a crate and connected through its backplane. The simplified real-time data path of the jFEX is shown in figure 2.5. The optical plant routes the data of the ECAL and HCAL to the jFEX subsystem. Each jFEX module has 5 optical receiver modules per processor FPGA. Each receiver module has 12 optical channels, summing up to 240 optical input fibers in total per jFEX module. The optical receiver modules are routed to high-performance Multi-Gigabit Transceivers (MGTs) of the processor FPGAs. These MGTs will operate at transfer rates of up to $12.8\,\mathrm{Gbit/s}$, while lower rates of $11.2\,\mathrm{Gbit/s}$ and $9.6\,\mathrm{Gbit/s}$ will be considered as well [8].

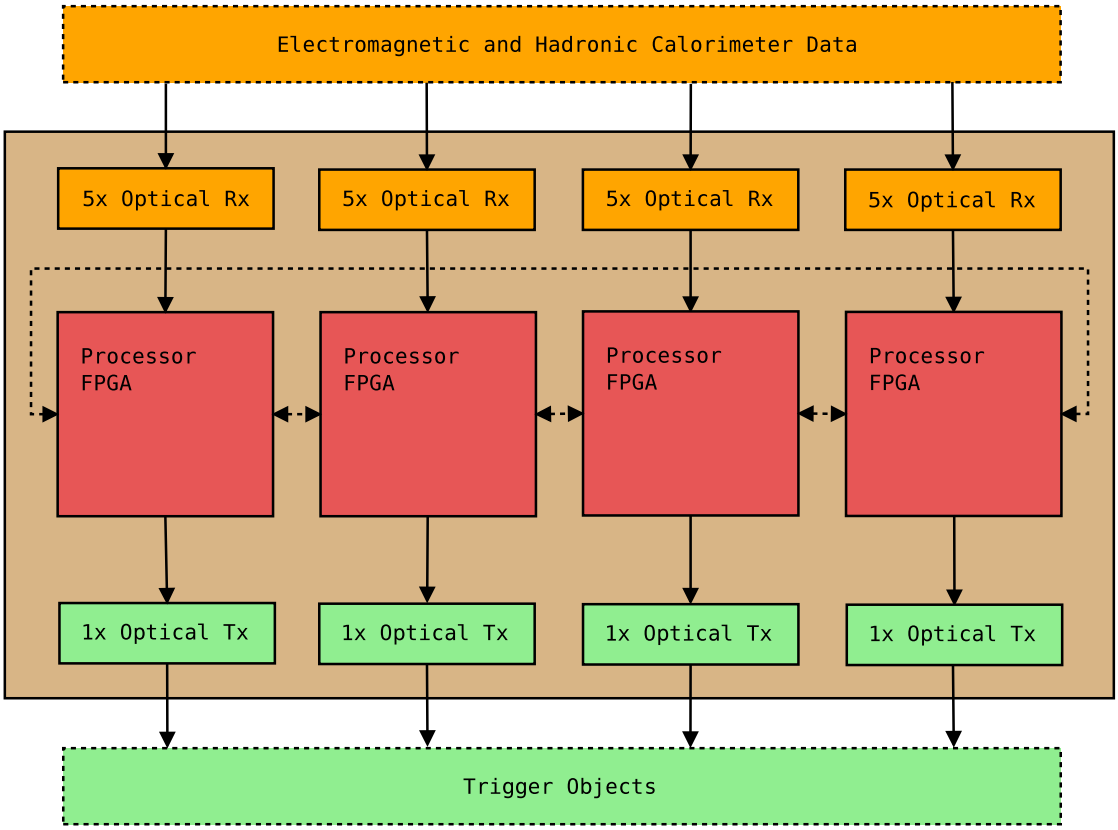


Figure 2.5: Real-Time Data Path of the jFEX

Furthermore, the MGTs are capable of simultaneously looping incoming data back [9]. The resulting duplicated data is routed to neighboring processor FPGAs forming a shared data ring between them as visualized by the dotted arrows in figure 2.5. This intra-module redundancy complements the inter-module redundancy used by some versions of the sliding window algorithm. The resulting TOBs of the processed data are transferred to the L1Topo. Since this will take less bandwidth, only a single optical transmitter module per processor FPGA is required, summing up to 48 optical output fibers per jFEX module. Additionally, each jFEX module provides the read-out connectivity to the RODs via MGTs through the backplane. Furthermore, a daughter module, here referred to as mezzanine card, provides a central control interface through the backplane in order to control the jFEX and all its subcomponents like a clock generator, the optical transceivers, and several power modules.

Chapter 3

Hardware Design

The module control of the jFEX of section §2.3 is implemented by a daughter module, a so-called mezzanine card. Its hardware design has been done in collaboration with Reinhold Degele who did the schematics, component placement, wire routing, and soldering supervised by Dr. Ulrich Schäfer. I did the conceptual design while complying to the given design specification, chose and ordered the electrical components including a further daughter module, designed its power sequencing, and created a net list to interface it.

3.1 Design Specification

Besides having algorithms running on processor FPGAs[1] of the jFEX to make time critical trigger decisions based on calorimeter data received over the real-time data path, other tasks and subcomponents have to be controlled and monitored as well. This is done by a central control FPGA. For reasons of flexibility this control FPGA is placed on a mezzanine card pluggable to the jFEX main board. Firstly, these algorithms must be programmed to the processor FPGAs at each power cycling. The implementation of an algorithm is stored in a so-called bit stream file. The expected file sizes are about $80\,\mathrm{MB}$ per processor FPGA, summing up to roughly $320\,\mathrm{MB}$ in total for all four FPGAs. Thus, a local storage which can be kept up to date by remote updates seems appropriate. To program an FPGA, such a bit stream file must be

[1] Processor FPGAs are of either UltraScale VU190 or UltraScale+ XCVU9P from Xilinx [10].

transferred over a dedicated parallel bus. In order to not wait a quarter of an hour or more to power cycle the jFEX, the final transfer rate of loading the bit stream files from the internal storage and copying them over to the FPGAs should be several $\mathrm{MB/s}$. It is also desired to monitor the operational states of the algorithms and to tweak their behavior by adjusting some parameters at run time. This requires additional connectivity to all processor FPGAs. For this and especially for the real-time data path, high-performance Multi-Gigabit Transceivers (MGTs) of the FPGAs are used to meet the required transfer rates of up to $12.8\,\mathrm{Gbit/s}$. For such high transfer rates over long distances, optical fibers are used instead of electrical wires for external communication by routing the MGTs to optical transceiver modules. These modules can be controlled by the I²C bus [11]. Since the incoming amount of data is higher then processed output data, transfer rate requirements for receivers and transmitters are not necessarily equal. The ATLAS TDAQ system [4] requires communication to be synchronous. Therefore, an I²C-programmable clock generator is used to derive synchronous clocks of different frequencies from a global input clock. In addition, this clock generator is capable of reducing possible jitter[2] of the input clock. Furthermore, the I²C bus is used to monitor various supply voltages of the power modules.

3.1.1 Central Control Interface

Due to its ubiquitous nature and therefore proven reliability and availability, Ethernet was chosen for the central control interface. Commonly used communication protocols are the Transmission Control Protocol (TCP) [12] and the User Datagram Protocol (UDP) [13]. While former guaranties in-order packet delivery by detecting packet losses and managing retransmissions itself, its implementation for an FPGA would be of a more complex state machine compared with latter protocol combined with sequential packet IDs for dropped packet recovery. Hence, an UDP based implementation called IPBus [14] was developed by the Bristol University and the Imperial College London, originally for the Triggering and Data Acquisition System (TriDAS) of the CMS experiment. It was agreed to be the standard communication protocol for controlling FPGA based hardware within the ATLAS TDAQ system [4].

[2] Jitter is an undesired deviation in periodicity of communication signals.

3.1.2 Intra-Board Communication

For controlling and monitoring the processor FPGAs, their built-in Multi-Gigabit Transceivers (MGTs) are used, allowing high transfer rates by using only six wires to each processor FPGA. Using differential signaling[3], one differential pair serves as clock while two further pairs allow full-duplex[4] serial communication.

In order to control and monitor several ICs[5] of the jFEX like the clock generator, the optical transceivers, and the power modules, the Inter-Integrated Circuit (I^2C) bus is used. It is a serial bus of two wires labeled SDA and SCL. Former transmits data synchronously to a clock of latter. It follows a master/slave model. Thus, a chain of ICs can be connected to the same two wires. Both wires are open-drain, that is they can only be driven low or left open. In latter case, resistors pull the wires high. Low and high are interpreted as logical “0” and logical “1”, respectively. A master starts a transaction to either read from or write to a slave, referred to as master read (MR) and master write (MW) mode. A slave serves these requests by writing or reading the data, referred to as slave write (SW) and slave read (SR) mode. A transaction is composed of an address byte and one or more data bytes transferred with the most significant bit (MSB) first, each followed by an acknowledge bit, “0” for acknowledging and “1” for not acknowledging. A byte is of $8\,\mathrm{bits}$. The address byte contains a 7-bit slave address followed by a direction bit, “0” for writing to and “1” for reading from a slave. When a slave detects its address, it is free to ignore a master request by not acknowledging the address byte, causing the master to abort the transaction. In MR mode the master acknowledges a byte to request the slave to write another one. Thus, each byte except the last is acknowledged. In contrast, in SR mode the slave acknowledges a byte to signify the master being able to read another one. Thus, each byte is acknowledged, but only in exceptional cases like a full data buffer a byte is not acknowledged, causing the master to abort the transaction. Additionally, one or more transactions are encapsulated by a frame of start and stop conditions while a repeated start condition separates multiple transactions within a frame. This allows multiple masters connected to the same bus by ensuring a master cannot start a transaction within a sequence of transactions of another master. A start condition is issued by firstly pulling the SDA wire low and then pulling the

[3] Differential signaling uses two electrically complementary signals.
[4] A full-duplex communication allows data transfer in both directions simultaneously.
[5] An integrated circuit (IC) is composed of electronic circuits on a compact chip.

SCL wire low while a stop condition is issued by firstly releasing the SDA wire and then releasing the SCL wire. Various speed modes are defined, among others, with standard mode of $100\,\mathrm{kbit/s}$ and fast mode of $400\,\mathrm{kbit/s}$.

3.1.3 Debugging Facilities

A universal asynchronous receiver/transmitter (UART) is a hardware module for serial communication commonly included in microcontrollers[6] and CPUs. Due to its simplicity it suits well for early stage debugging like a boot process of a CPU. A configurable number of data bits can be sent, usually eight bits, but five to nine bits are possible as well. If not nine bits are chosen, an optional parity bit[7] can be appended giving a chance of error detection. These data bits along with an optional parity bit are encapsulated by a start bit of value “0” and one or two stop bits of value “1”. A value of “0” or “1” is represented by a low or high signal, respectively. Thus, when idle, the signal is kept high to be able to detect a start bit. The frame of start and stop bits is required for synchronization since there is no clock signal. A voltage level translator [15] fulfills the electrical characteristics defined by the Recommended Standard 232 (RS-232) for serial communication commonly used for serial ports of computers. Alternatively, the Universal Serial Bus (USB) can be interfaced with the help of an onboard USB-to-UART bridge [16].

3.2 Conceptual Design

An FPGA allows customizable, true parallel, and accurately timed digital signal processing (DSP), making it a first-choice core component of hardware designs with focus on time critical signaling and long-term use. But tasks of less time accuracy like accessing an SD card[8] or processing data on a higher level to control modules like a clock generator are well suited to a CPU running an operating system with a whole ecosystem of applications and libraries. To use advantages of both worlds, a CPU and an FPGA, communication between them is required to delegate tasks or share data. Instead of using separate chips for CPU and FPGA with externally

[6] A microcontroller is a compact computer with I/O peripherals on a single IC.
[7] A parity bit indicates whether the number of data bits with value “1” is even or odd.
[8] A Secure Digital (SD) card is an exchangeable non-volatile data storage device.

hardwired connectivity, a more flexible alternative was chosen, a hybrid system on a chip (SoC), combining an FPGA and a CPU inside a single chip with many customizable interconnects in between. In regard to a short development time, the first iteration of the mezzanine card does not populate this hybrid SoC on its own, instead it is interfaced through a daughter system on a module (SoM) from Avnet called PicoZed 7Z030 [17] based on a hybrid SoC from Xilinx called Zynq XC7Z030 [18]. The PicoZed is plugged onto the mezzanine card, already fulfilling the electrical requirements of the Zynq. In figure 3.1 the block diagram of the mezzanine card is shown. The brown, dark green, and light green boxes represent a simplified jFEX main board, mezzanine card, and PicoZed daughter module, respectively while each is plugged onto the other. The purple box within the light green one represents the Zynq with its CPU in blue and its FPGA in red, referred to as processing system (PS) and programmable logic (PL), respectively. Components are colored blue or red whether they are chained with the CPU or the FPGA, respectively while components being chained with both or being connectible to both are colored purple like the Zynq.

Though the first iteration of the jFEX will have wire routings for all four processor FPGAs, it will be populated with only two of them to firstly gain experience with initial tests. Thus, only connectivity for two processor FPGAs is required for the first iteration of the mezzanine card. The PicoZed series was chosen over another SoM series from Avnet called MicroZed [19], since that series does not provide MGTs, which are required to control and monitor the processor FPGAs. Two of four available MGTs of the PicoZed are reserved for the processor FPGAs while a third one is connected to a PHY[9] for Ethernet communication via FPGA. This PHY is populated on the mezzanine card and exposed by an Ethernet jack also known as RJ45 connector. Another Ethernet jack is connected to a PHY of the PicoZed allowing Ethernet communication via CPU. This whole setup is supposed to be installed in a crate of multiple devices being interfaced through the crate backplane (BP). Among other connectivity, this backplane is connected through the jFEX to a third Ethernet jack in purple of the mezzanine card. Its purpose is to be either connected to the Ethernet jack of the FPGA in red or to the Ethernet jack of the CPU in blue. In this way, as a precautionary measure regarding possible future changes to the central control interface of section 3.1.1, it is possible to choose between FPGA based or CPU based control through the backplane by using an Ethernet cable instead of multiple $0\,\Omega$

[9] A physical layer device (PHY) implements the physical layer of a communication protocol.

resistor jumpers on the mezzanine card. In any case, the Ethernet jack of the CPU is used as development and maintenance interface.

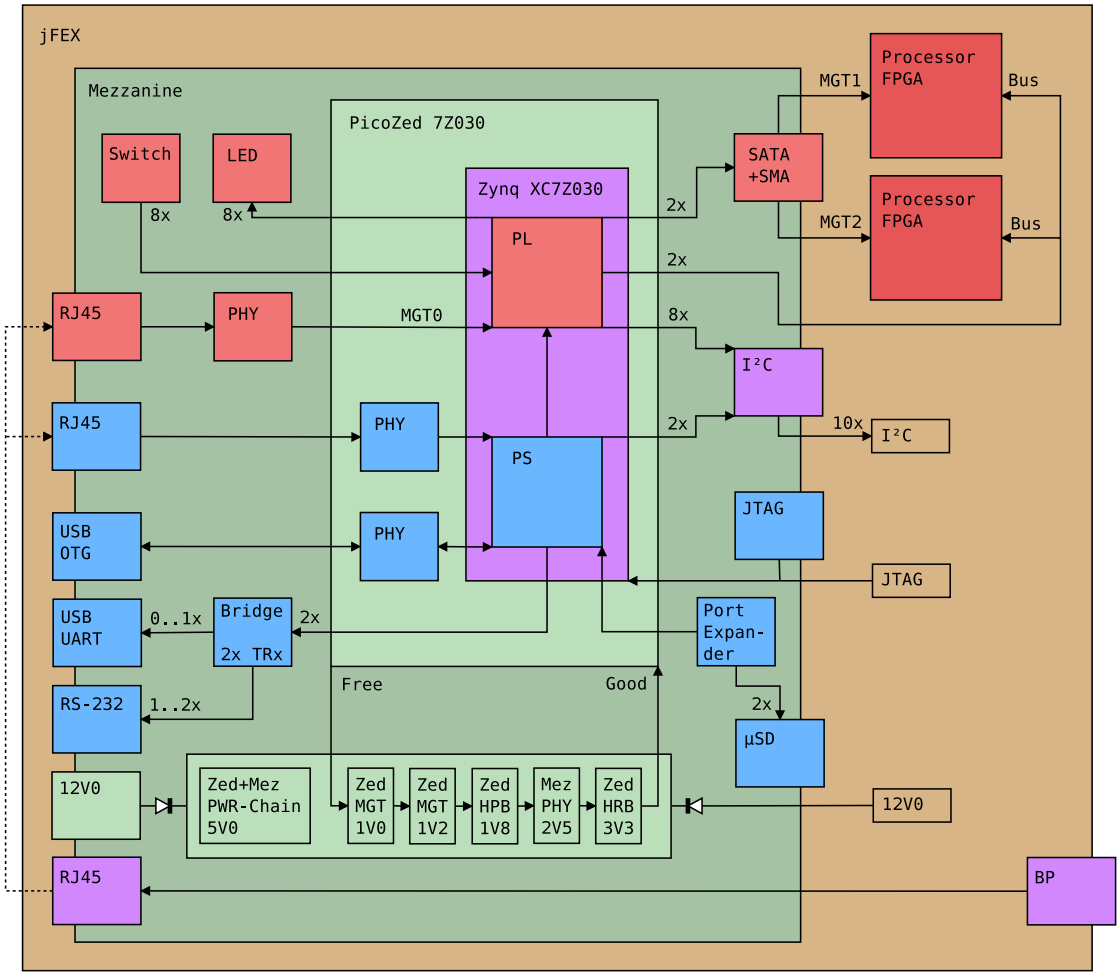


Figure 3.1: Block Diagram of the Mezzanine Card

Though the mezzanine card is supposed to be plugged onto the jFEX, it can operate without it as well, allowing standalone tests. Thus, it is powered by either the jFEX or an external power supply connected to its power jack. Both $12\,\mathrm{V}$ supply voltages, the one of the jFEX and the one of the external power supply, are connected to a chain of further power supplies [20] of the mezzanine card represented by the light green box in figure 3.1, each regulating a different supply voltage. Two diodes protect the $12\,\mathrm{V}$ power supplies from each other in case both are connected at the same time. Each power supply of the chain can be enabled by raising its input pin labeled “EN”. It will indicate when its power is good (PG), that is its voltage has stabilized, by raising an output pin labeled “PG”. When power cycling the mezzanine card, the

PicoZed is permanently supplied with $5\,\mathrm{V}$. But supplying its banks[10] is required to be delayed until the internally regulated supply voltages of the PicoZed have stabilized. Furthermore, it is advised to supply MGT voltages in a sequence from low to high voltages to reduce current drawn during power up [21]. As visualized in figure 3.1, the PicoZed exposes a signal labeled “Free”, indicating when the mezzanine card is free, in a sense of being allowed, to supply the banks. This signal is connected to the input pin “EN” of the first power supply of the chain, enabling one power supply after another by cascading the output pins “PG” of each power supply to the input pin “EN” of the next power supply in the sequence they are supposed to power up. The output pin “PG” of the last power supply of the chain is connected to the signal of the PicoZed labeled “Good” to finally enable the Zynq after all supply voltages have been stabilized, which are $1.0\,\mathrm{V}$ for MGT transceivers, $1.2\,\mathrm{V}$ for MGT termination circuits, $1.8\,\mathrm{V}$ for high-performance banks[11], $2.5\,\mathrm{V}$ for the PHY of the mezzanine card, and $3.3\,\mathrm{V}$ for high-range banks[12] in the order mentioned.

The Zynq boots from one of two SD cards via an SDIO port expander [22] serving as voltage level translator as well. The SD card is selected by two jumpers, the second SD card serves as fallback system. Since the PicoZed provides an USB On-The-Go (OTG) interface, an appropriate micro-AB USB plug was populated on the mezzanine card. The two UART modules of the Zynq are connected to a RS-232 voltage level translator, each exposed by a header[13] of three pins, two for receiver (Rx) and transmitter (Tx) plus one for ground (GND). Additionally, one of these UART modules can be connected to a USB-to-UART bridge via $0\,\Omega$ resistor jumpers, exposed by an micro-B USB plug. As further debugging facilities, an array of eight switches and an array of eight LEDs, connected to the FPGA, can be used as quick configuration and status indicator, respectively.

All connectivity of the header connector[14] to the jFEX is additionally exposed on separate headers and connectors to allow standalone tests of the mezzanine card. Two I²C modules of the CPU and eight I²C modules possibly implemented by the FPGA are exposed by an array of headers of three pins each, two for SDA and SCL plus one for GND. Each of the two MGTs for the processor FPGAs can be accessed

[10] A bank of an FPGA is a group of I/O pins sharing common resources like a supply voltage.
[11] The PicoZed exposes 100 high-performance (HP) [23] user PL I/O pins.
[12] The PicoZed exposes 35 high-range (HR) [23] user PL I/O pins.
[13] A pin header or simply header is of one or more rows of pins.
[14] A header connector is meant to be plugged onto a header.

by a SATA[15] connector when configuring their $0\,\Omega$ resistor jumpers appropriately, allowing a loop back test with a crossover SATA cable. The differential MGT reference clock is available by two SMA connectors[16]. The JTAG[17] debug interface of the Zynq, cascaded with the processor FPGAs of the jFEX to be part of a boundary scan, is exposed by a separate header as well.

3.3 First Iteration

The first iteration of the mezzanine card is shown in figure 3.2. Its two SD card slots are on the right above the Ethernet jack of the CPU. On the left of this jack are seven of the eight I²C headers of the FPGA. The eighth one was placed more to the left near the JTAG header. The two I²C headers of the CPU are below the two golden SMA connectors for the differential MGT reference clock. The SATA connectors of the two MGTs are on the left below the power jack and above the switch and LED arrays. The Ethernet jack of the backplane is on the left bottom. The USB cable on the top is connected to the USB-to-UART bridge with the USB OTG plug on its left and the Ethernet jack of the FPGA on its right. The PicoZed can be plugged onto the three white headers.

[15] Serial AT Attachment (SATA) [24] is a computer bus interfacing mass storage devices.
[16] SubMiniature version A (SMA) connectors are commonly used for coaxial cables.
[17] The Joint Test Action Group (JTAG) [25] interface allows testing of IC interconnects.

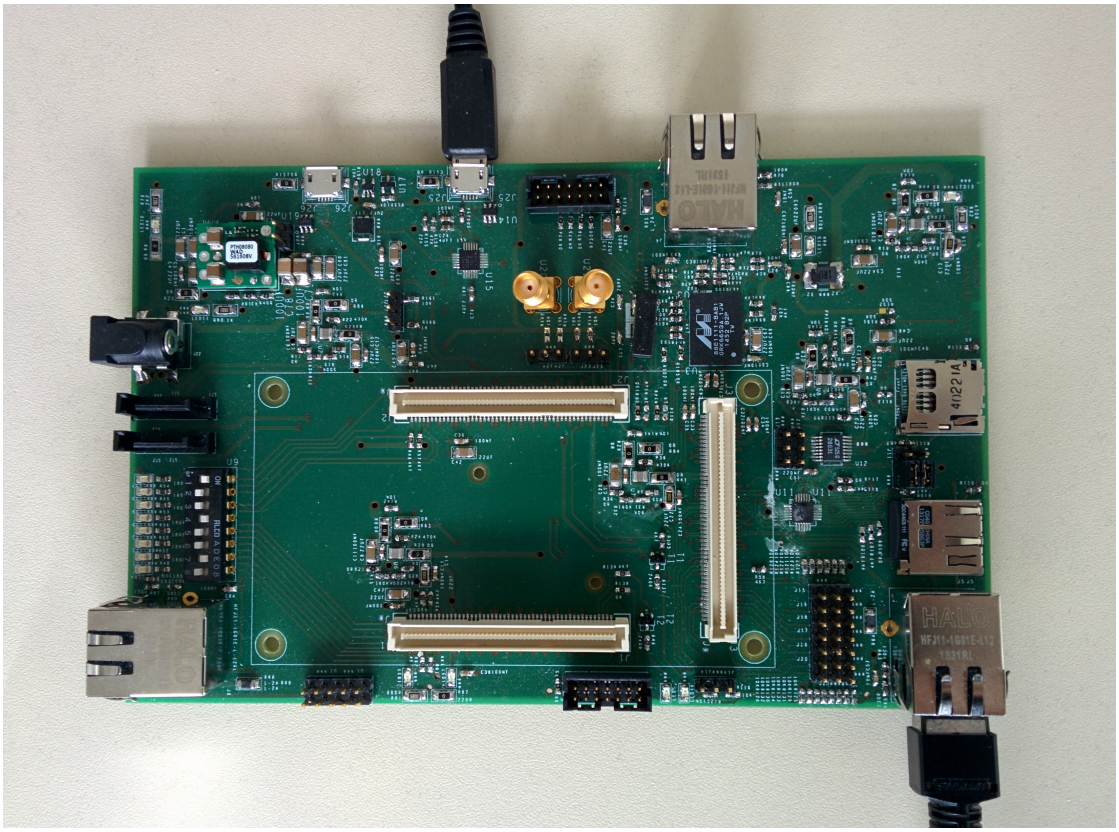


Figure 3.2: Mezzanine Card

At the time of finishing the hardware design of the mezzanine card, the pin layout of the header of the jFEX, to connect the mezzanine card onto, was not yet finalized. To not postpone the production and debugging of the mezzanine card, it was decided to drop its header connector, but for the first iteration only.

Chapter 4

Software Development

In order to test, control, and advance the mezzanine card of section §3.3, several software packages have been developed for long-term use including a workflow kit named Zed Tool (`z2l`), a helper tool polling for pluggable devices named `plug`, a file transfer application between CPU and FPGA named Feedback Synchronized I/O (`fsio`), and a controller of the clock generator named `si53xx`. These software packages including their sources are provided on a compact disk (CD) attached to this thesis. Additionally, each software package is hosted on a web server at following addresses:

`https://qu1x.org/z2l`
`https://qu1x.org/plug`
`https://qu1x.org/fsio`
`https://qu1x.org/si53xx`

They have been developed with focus on reusability and extendibility in mind by structuring their source code hierarchically from abstract interfaces down to concrete implementations while not sacrificing performance to allow flexible and extensible use cases. Finally, libraries are served with general-purpose applications to ease use of them while these applications and libraries along with manual pages and quick introductions are distributed in a package format allowing automated configuration and compilation with the help of the GNU Build System also known as the Autotools [26]. Compact algorithms and interfaces, only parts of the source code with a so-called high expressiveness, are shown, allowing their motivational description to reason and explain the conceptional choices of the software design in a whole, being an essential part of this thesis, and to finally serve as documenta-

tion for further development. Software was developed with Debian Jessie (amd64[1]) [27] as operating system and was additionally tested with PetaLinux 2015.4 (arm-linux-gnueabi[2]) [28] from Xilinx, an embedded operating system running on the mezzanine card.

4.1 Workflow Kit

Development for the hybrid SoC Zynq is composed of three major processes. The configuration of an FPGA is represented by a bit stream file described by a hardware abstraction language (HDL). This bit stream file can be generated by the Vivado Design Suite [29] from Xilinx. The CPU runs an embedded operating system called PetaLinux. It can be customized and compiled with the PetaLinux Tools from Xilinx. On top of this operating system, user application software is installed. These applications are cross-compiled with the help of the GNU Build System also known as the Autotools. All these three toolchains, the Vivado Design Suite, the PetaLinux Tools, and the Autotools store their configurations, source files, and destination files in their own project folders. While cross-compiling application software is almost independent from configurations of the other two projects, the PetaLinux Tools require configurations to be imported from the Vivado Design Suite project in order to appropriately customize the operating system. To manage all three projects and automate their integration into each other to speed up the development process, the software package Zed Tool (`z2l`) has been developed.

4.1.1 Dependency Resolution

Zed Tool combines all three projects into a single higher-level project. It resolves dependencies between the three toolchains by automatically invoking them to generate their destination files from their source files, like the bit stream from the HDL sources, a boot image and a root file system image of the operating system from its source code and configuration files, and the application binaries from their source code files. Such automation is possible thanks to Vivado being based on a scripting

[1] The AMD64 port supports the 64-bit version of the x86 instruction set.
[2] The arm-linux-gnueabi cross-compiler target without hard-float support is used.

language called Tool Command Language (TCL) [30] allowing it to be invoked without its graphical user interface (GUI). The three projects along with their toolchains, Vivado Design Suite, PetaLinux Tools, and Autotools are referred to as hardware (HW), firmware (FW), and software (SW) projects, respectively. Their own dependencies between destination and source files and their inter-project dependencies are visualized in figure 4.1. Zed Tool automatically resolves these dependencies by comparing timestamps of destination files with the ones of their source files. When a destination file is older than one of its source files, it will be regenerated by invoking its appropriate tool. A common tool for resolving such file dependencies is called GNU Make [31]. The dependencies are described by a so-called Makefile which is parsed by this tool. Alternatively, a self-executable Makefile automatically invokes Make when it will be executed, which is the case for Zed Tool. The three dependency stages, HW in blue, FW in green, and SW in orange of figure 4.1 are represented by their destination files in dark blue, dark green, and dark orange, respectively while their sources are lightly colored. The normal green colored components compose the dark green ones, indicated by having no gap between them. Continuous arrows are intra-project dependencies while dashed arrows are inter-project dependencies, these are destinations being sources of other projects. While the SW stage is simply represented by application binaries without inter-project dependencies, the HW stage is made up of two destination files, a bit stream file and a hardware description file, not to be confused with the HDL files. Both are sources of the FW stage.

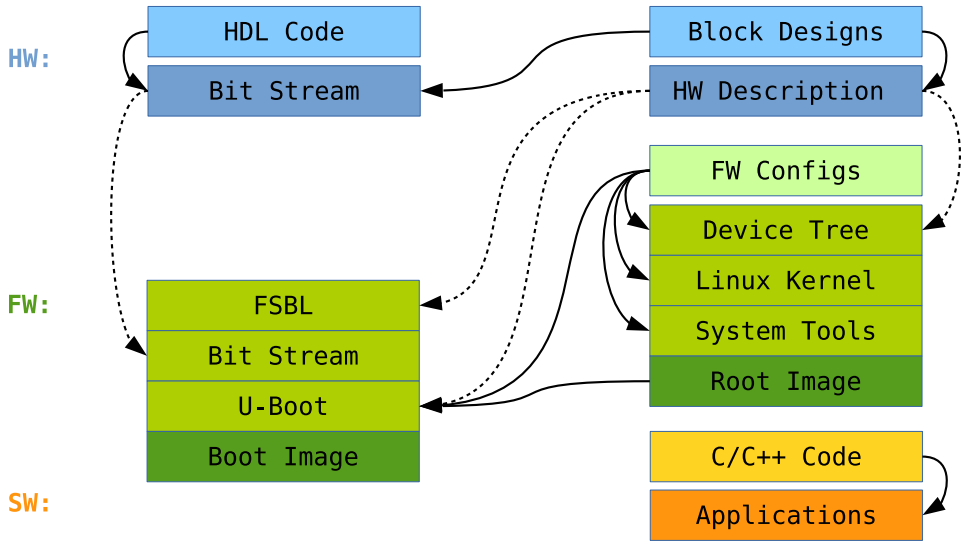


Figure 4.1: Dependency Graph

While the bit stream depends on both HDL and block design files, the hardware description file is independent of HDL files. Block designs can describe interconnects between CPU and FPGA and are used to configure hardware interfaces like I²C or UART modules. They are made up of Intellectual Property Cores (IP Cores) which can be instantiated by selecting them from the Vivado IP Catalog. A remarkable property of this dependency graph is that there is no single dependency from left to right, arrows only point from right to left. Thus, when only HDL files change, it is not necessary to invoke the tools for regenerating the root image, which would actually invoke a lot of other internal Makefiles of the PetaLinux Tools to recheck all destinations for their possibly updated sources, which are the first stage boot loader (FSBL) [32], the Universal Boot Loader (U-Boot) [33] also known as "Das U-Boot", the device tree [34], the Linux kernel [35], and all the system tools included in the root file system, which requires time. The FSBL and the U-Boot are actually components of the boot image but they are generated by the root image generation process and just copied over when generating the boot image. The FSBL in conjunction with the U-Boot make up a two stage boot loader to load the kernel which parses the device tree providing information about available hardware and their configurations. Finally, the kernel mounts the root file system and starts several system services. The problem is, when Vivado is invoked to regenerate the bit stream due to changed HDL files, it additionally regenerates the hardware description file though no block design files have changed. This would unnecessarily trigger the regeneration of the root image. To prevent this, the hardware description file is duplicated by Zed Tool but only when block design files have changed. This duplicate is used as HW destination instead of the original one which is unnecessarily regenerated each time HDL files have changed. In this way, the development process is speed up by saving the time of the root image generation process each time HDL files but no block design files have changed. After regenerating the bit stream due to changed HDL files, only the boot image generation process is triggered which is of almost no time compared with the root image generation process.

4.1.2 Modification Management

Besides improvements regarding development time by focusing on the sources themselves instead of remembering how and when to regenerate their destinations, another goal of Zed Tool is to make image generation reproducible which is indispensable for long-term use and maintenance. It allows to distribute the higher-level project as source files only, being lightweight compared with the final size of a whole operating system. The Vivado Design Suite already allows separation of source and

destination folders while the Autotools have routines to clean up the project folder from its destination files. Though the PetaLinux Tools have similar clean up routines, there are still a lot of generic files left. Thus, Zed Tool manages saving, loading, editing, and diffing[3] of essential configuration files of PetaLinux and its various subsystems, which are the FSBL, the U-Boot, the device tree, the Linux kernel, and the system tools to be included in the root file system. Since Zed Tool is a self-executable Makefile, it accepts only targets, besides options for Make itself, as command line arguments, which are nothing else than destination files or aliases of destination files. It is also possible to define so-called phony targets not being related to a file in order to implement commands. Zed Tool provides a GNU Bash[4] completion script as well. When writing `z2l` in a terminal prompt followed by double pressing `TAB`, the terminal will suggest all available alias targets for Zed Tool, as shown in figure 4.2. The five targets `fw.edit.plt`, `fw.edit.ubl`, `fw.edit.dts`, `fw.edit.lnx`, `fw.edit.rfs` allow editing the FSBL, the U-Boot, the device tree, the Linux kernel, and the system tools to be included in the root file system, respectively while `fw.edit.sys` allows to edit the configuration of PetaLinux itself and `fw.edit` invokes all previously mentioned targets one after the other. Changes made, are directly stored in the project folder of the PetaLinux Tools. After confirming their functionality by testing these modifications, they can be saved to a separate PetaLinux source folder created by Zed Tool by invoking `fw.save`. In case these modifications caused more problems instead of reducing them, they can be revoked by loading the previous configurations of the source folder with `fw.load`. When a lot of modifications have been done at once, the target `fw.diff` gives an overview of them by only printing the differences of configurations in the destination folder compared with the ones in the source folder, allowing to confirm them before possibly saving them with `fw.save`. In contrast, HW modifications can be done by launching Vivado with its GUI and letting it automatically open the HW project by invoking the target `hw.edit`.

[3] Comparing two files with a common tool called diff [36].
[4] GNU Bash [37] is the default Unix shell of Debian for interactive terminals.

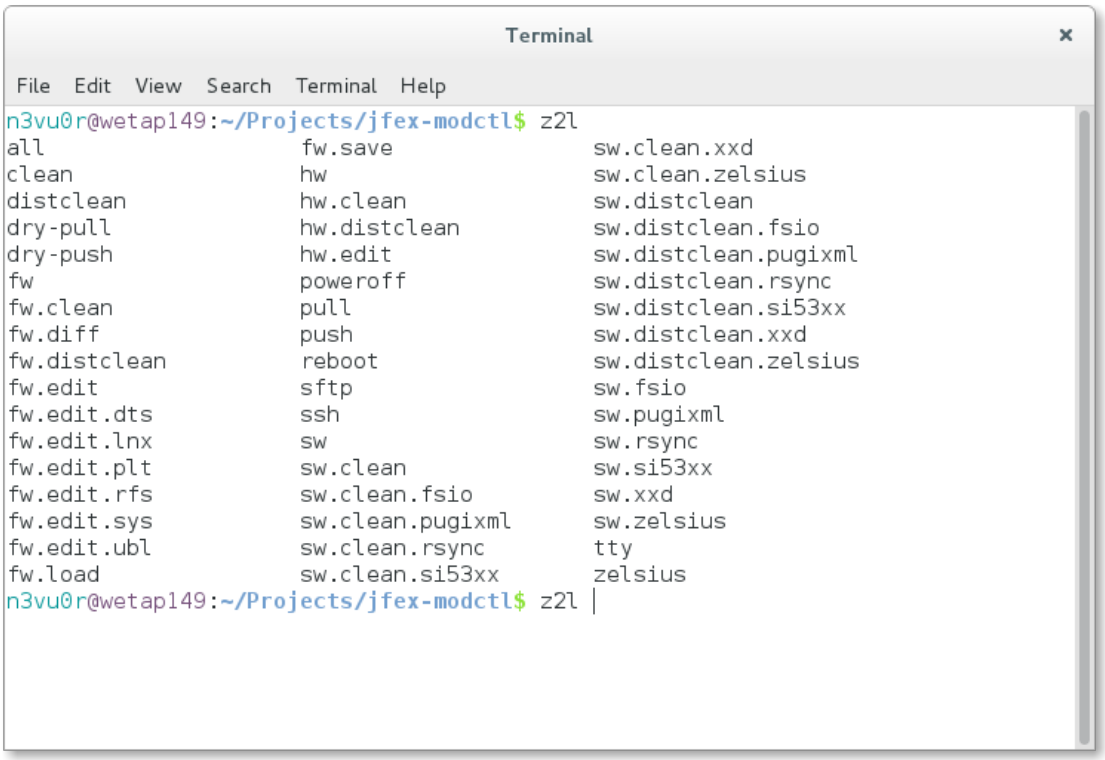


Figure 4.2: Usage of Zed Tool

4.1.3 Development Process

To actually test modifications, not only those to PetaLinux configurations but also modifications to HW or SW projects, the appropriate modified destination files have to be transferred to the target device, the mezzanine card. What can cause a lot of trouble is when modifications have been done but some modified destination files have been forgotten to be transferred. This gives the impression that these modifications either have no effect since they have actually not been applied or break the system since they have actually been applied only partially. Zed Tool keeps track of all modified destination files with the help of a common tool called `rsync` [38]. It allows local or remote updates by synchronizing a destination folder regarding its source folder by a technique called rolling checksum. Additionally, it provides a so-called dry-run mode simulating a synchronization to confirm nothing unintended will accidentally happen. Thus, to automatically transfer all modified destination files or to only simulate it, the targets `push` or `dry-push` are used, respectively. For doing the opposite, that is making a backup of the target device, the targets `pull` and `dry-pull` are available. In case the bit stream was modified, a reboot is required to reprogram the FPGA, which can be triggered with the `reboot` target. In order to

monitor the boot process, an UART or an UART over USB terminal connection can be establish with a common terminal emulator called `picocom` by invoking the `tty` target. After the device has booted, a more reliable TCP terminal connection can be established with the help of a common tool called Secure Shell (SSH) [39] by invoking the `ssh` target. All this can be done at once by passing these targets to Zed Tool from left to right in the order they should be invoked with a command line prompt of `z2l push reboot tty ssh`. There is one problem, after triggering a reboot, the UART device vanishes causing the `tty` target to fail. To solve this, a helper script was written polling for the UART device until it is available again. Due to its general-purpose usability this script is separately distributed as the software package `plug`. When the boot process has finished, the terminal can be exited by pressing `CTRL+A+X`. Afterwards, the SSH terminal is established for testing the modified applications. Besides synchronization of the whole destination folder, partial transfers can be done with the `sftp` target as well, making use of the SSH File Transfer Protocol. On-the-fly modifications of configuration files at the target device are also possible over this protocol. Thanks to the default file browser of Debian called Nautilus, these remote files can transparently be accessed like local files. The destination files of the three dependency stages HW, FW, and SW have corresponding alias targets `hw`, `fw`, and `sw`, respectively. First one is practical to test if modifications to HDL and block design files are valid without triggering the regeneration of the root image when the intention is to continue modifying these HW sources. The targets `sw.PKG`, `sw.clean.PKG`, and `sw.distclean.PKG` behave similarly to the standard Make targets `all`, `clean`, and `distclean` of the software package labeled “PKG”, respectively. The targets `sw`, `sw.clean`, and `sw.distclean` invoke the corresponding targets of all available software packages. The first of these, `sw.PKG` or `sw`, additionally installs the software package or all software packages to the local destination folder, respectively. It can then be synchronized with the remote destination folder. Another helper script, invoked with target `zelsius` but in contrast to `plug` being distributed with `z2l`, reads out the raw temperature data of the Zynq analog-to-digital converter (XADC) and prints the calculated SoC temperature in millidegree Celsius on the screen.

4.1.4 Toolchain Encapsulation

Further handy features of Zed Tool are encapsulating environment settings, required to be loaded before the Vivado Design Suite and the PetaLinux Tools can be executed, and to serve the addresses of license servers and paths to license files, required for certain features or IP Cores of the Vivado Design Suite. This encapsulation is necessary, since otherwise these settings would break functionality of

other tools. For instance, the order in which library paths are scanned is changed to make PetaLinux Tools work, but causes subsequent invocations of other tools to fail. Thus, for each executable, Zed Tool installs a so-called wrapper script launching a sub-process and changing only the environment of this sub-process, which then invokes the corresponding executable. When terminated, the sub-process will be terminated as well, but the environment settings of the main process, the actual terminal, were not being touched.

4.2 CPU/FPGA Communication

In order to use the advantages of both a CPU and an FPGA, communication between them is required to delegate tasks or share data. The so-called physical address space of the CPU has a reserved address range to access dedicated registers of the FPGA. Since the CPU runs an operating system whose kernel uses a memory management technique called virtual memory [40], only kernel modules have direct access to the physical memory while application software must map physical addresses to their virtual address space. This mapping is done by the system call[5] `mmap` [41]. It takes a physical address as argument and returns an appropriate virtual address used by the software application. When accessing the virtual address, it is translated to the physical address and its data might be cached by the kernel, resulting in a possibly delayed, so-called indirect memory access. Thus, there are two ways of establishing a communication channel, developing either a kernel module or an application software library managing the memory mapping. While a kernel module might promise higher transfer rates and less latency due to its direct memory access (DMA), interfacing the DMA API[6] [42] of the kernel is a more complex task requiring more parameters to be studied, resulting in longer development time and increased maintenance effort in case of new kernel versions. In contrast, setting up memory mapped channels is more flexible requiring no additional parameters among the address ranges to be mapped, each specified by its base address and width. For these reasons, implementing memory mapping by an application software library was chosen. Thus, the software package `fsio` has been developed.

[5] A system call allows application software to request a certain service from the kernel.
[6] An application programming interface (API) provides routines to use a software component.

Along with its library `libfsio`, it contains two binaries, `fsio` and `fsio-tvgen`. Former is a file transfer application while latter is a test vector[7] generator.

4.2.1 Communication Protocol

Due to the nature of caching and clock domain crossing between CPU and FPGA, reading data from and writing data to the FPGA must be synchronized by an out-of-band[8] handshake[9] signal to avoid data corruption. For example, writing data from the CPU to the FPGA must be finished before the FPGA reads it, otherwise the data might be partially out of date. Thus, the CPU has to inform the FPGA when it is allowed to read the data by toggling a separate handshake signal after the data has been written. But just executing one write instruction after another is not enough, since caching does not guarantee to strictly preserve their order. A solution to this problem is, to continuously read the written data back on a separate mapping and compare them until they are equal, before executing another instruction. Hence the name Feedback Synchronized I/O (`fsio`) was chosen for this software package.

To make the communication stateless regarding the actual value of the handshake signal, a master/slave model was chosen with the CPU as master and the FPGA as slave. That means, only the CPU is allowed to toggle the handshake signal. In this way, an application does not need to remember the previous value each time it will be executed. Instead, it can just read it from the FPGA after mapping a channel without otherwise potentially missing a handshake toggle. Thus, there are four routines, master write (MW), slave read (SR), master read (MR), and slave write (SW). First two are used to write to the FPGA while last two are used to read from the FPGA, as illustrated in figure 4.3 and 4.4, respectively. In both figures, the CPU on the left and the FPGA on the right represent an unidirectional communication channel. The arrows indicate the two feedback loops of each figure, one for the data and one for the handshake. A loop has both input and output mappings, labeled “fsi” and “fso” for the data, and “hsi” and “hso” for the handshake, respectively. What is labeled as input for the CPU, is labeled as output for the FPGA, and vice versa. Time flows from top to bottom, except for the orange colored feeding which is done continuously on the FPGA and alternately with the handshake polling on the CPU.

[7] A test vector is made up of specific input data to test a given software component.
[8] An out-of-band signal exchanges information in a separate channel from the data stream.
[9] A handshake allows requesting and acknowledging chunks of data between two entities.

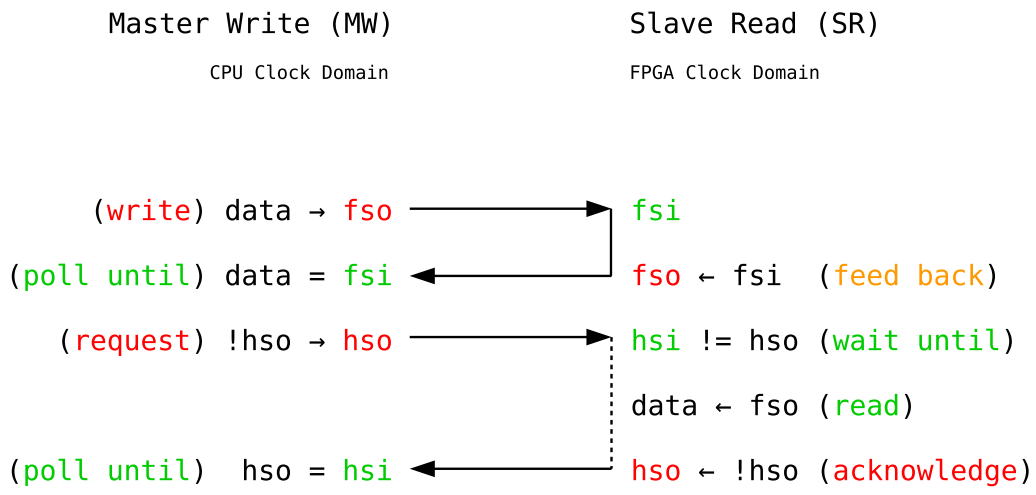


Figure 4.3: Writing To the FPGA

The master write routine firstly writes the data, and repeatedly reads it back until they are equal. Then it toggles the handshake signal requesting the FPGA to read the data. Afterwards, the slave read routine of the FPGA acknowledges the request by feeding back the toggled handshake signal for which the CPU is waiting for by polling it.

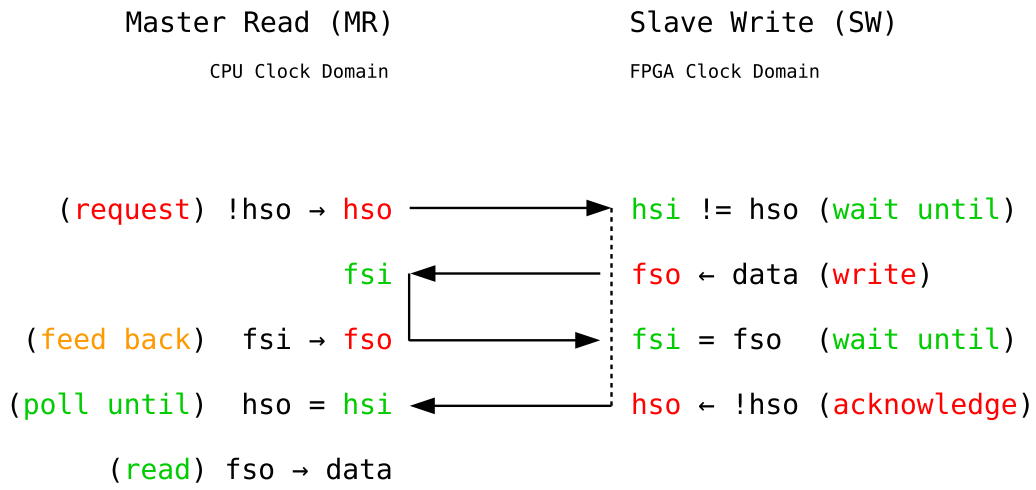


Figure 4.4: Reading From the FPGA

In contrast, the master read routine firstly toggles the handshake signal requesting the FPGA to write data while the CPU repeatedly feeds it back. The slave write routine of the FPGA compares the written with the fed back data until they are equal. Afterwards, it acknowledges the request by feeding back the toggled handshake signal, allowing the CPU to finally read the data.

4.2.2 Memory Mapping

The software tool used to create FPGA designs is the Vivado Design Suite from Xilinx. Interconnects from FPGA to CPU are described by a so-called block design, as shown in figure 4.5. The block “processing_system7_0” represents a reduced interface of the CPU while the registers to be mapped are represented by the blocks “axi_gpio_0”, “axi_gpio_1”, “axi_gpio_2”, “axi_gpio_3”, and “axi_gpio_4”, here referred to as AXI GPIO maps. Blocks are Intellectual Property Cores (IP Cores). They can be instantiated by selecting them from the Vivado IP Catalog. The five AXI GPIO maps are instances of the AXI GPIO IP Core[10], connected to the CPU via the Advanced eXtensible Interface (AXI) of the Advanced Microcontroller Bus Architecture (AMBA). While the CPU acts as AXI master, the five maps act as AXI slaves. The block “processing_system7_0_axi_periph” between them manages memory mapped transfers of one or more masters to one or more slaves.

[10] The AXI GPIO IP Core [43] allows general-purpose I/O (GPIO) CPU/FPGA interconnectivity.

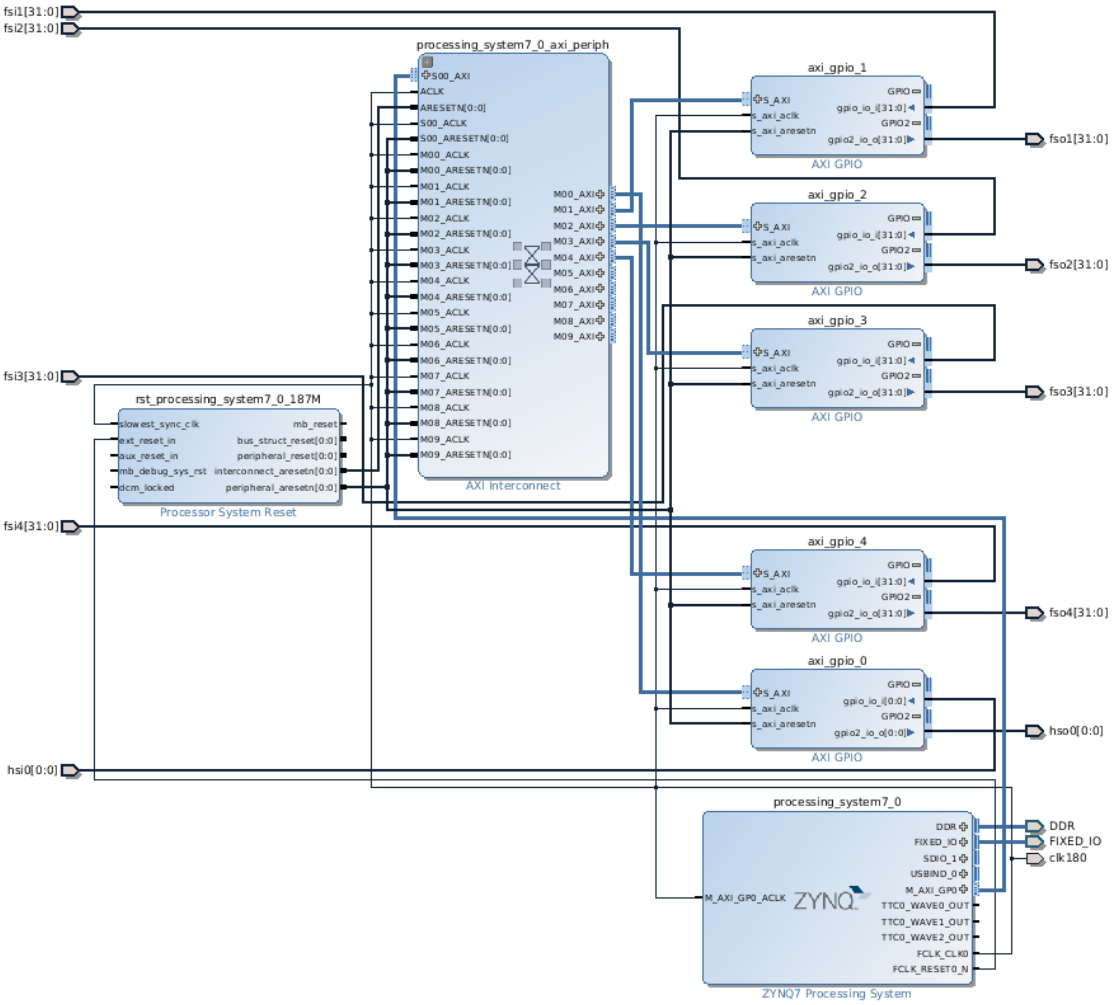


Figure 4.5: CPU/FPGA Interconnects [29]

As shown in figure 4.6, a map provides a so-called dual channel mode. This allows the first channel to be input only and the second channel to be output only. This suits the proposed protocol well since one of the two channels can be used as feedback loop reducing the number of maps to not bloat the block design.

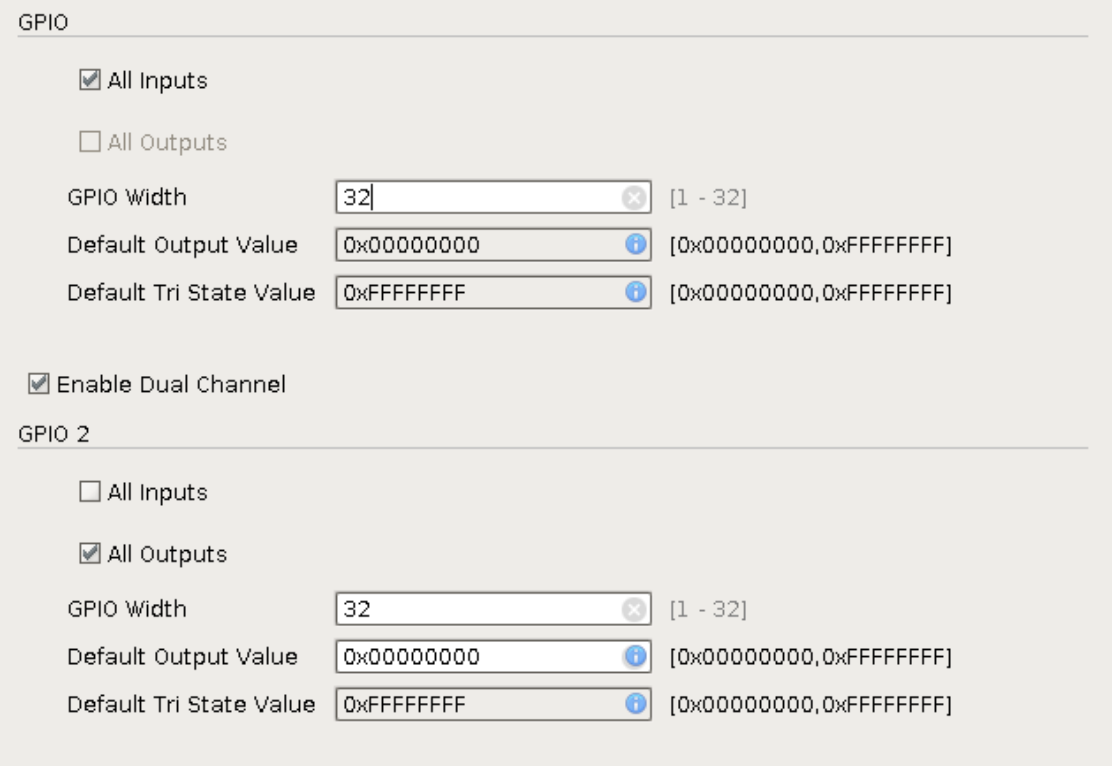
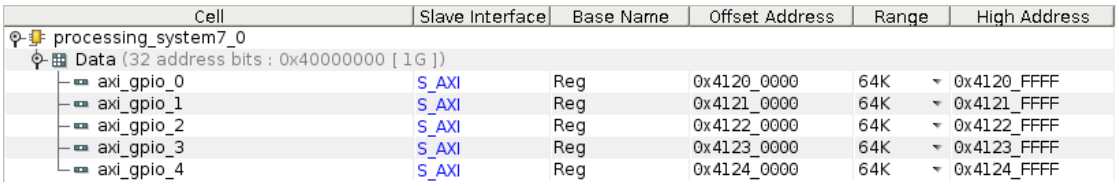


Figure 4.6: Map Customization [29]

The first of the five maps is the handshake map with a single input and a single output signal labeled “hsi” and “hso” while the other maps for the actual data are made up of 32 signals each, labeled “fsi” and “fso” followed by the instance number, again for input and output, respectively. A map is of a maximum width of $4\,\mathrm{B}$. Synchronizing multiple of these data maps at once, would ideally increase the throughput rate by a factor of the map count. But a CPU can execute instructions only sequentially, apart from some exceptions. However, the reduced number of handshakes would still increase the throughput rate up to a certain saturation. Besides this, there might be another effect. Due to the delay of indirect memory access, these sequential instructions might effectively appear parallel on the FPGA in case the time difference between them is smaller. The delay might neither be constant nor precisely predictable due to complex underlying kernel architectures like caching mechanisms. But in principle the throughput rate might increase the more maps are synchronized at once, until the time differences of instructions sum up to the delay, resulting in saturation. Thus, this design contains multiple data maps for a single handshake map, here referred to as a communication channel of widths $4 \cdot 4\,\mathrm{B} = 16\,\mathrm{B}$. Though it has input and output mappings, it is unidirectional since one direction is used as feedback only.

After composing a channel, that is instantiating one handshake and one or more data maps, the base addresses of the single maps are displayed in the “Address Editor” tab of Vivado, labeled “Offset Address” as shown in figure 4.7.



Cell	Slave Interface	Base Name	Offset Address	Range	High Address
processing_system7_0					
Data (32 address bits : 0x40000000 [1G])					
axi_gpio_0	S_AXI	Reg	0x4120_0000	64K	0x4120_FFFF
axi_gpio_1	S_AXI	Reg	0x4121_0000	64K	0x4121_FFFF
axi_gpio_2	S_AXI	Reg	0x4122_0000	64K	0x4122_FFFF
axi_gpio_3	S_AXI	Reg	0x4123_0000	64K	0x4123_FFFF
axi_gpio_4	S_AXI	Reg	0x4124_0000	64K	0x4124_FFFF

Figure 4.7: Map Addresses [29]

In algorithm 4.1, these addresses along with the widths of the channel maps make up the channel description file intended to be parsed by the library `libfsio`. To access the second channel of a dual channel map, an offset of 8 B is added to the base address of the first channel as described in the product guide [43] of the AXI GPIO IP Core.

Algorithm 4.1 Channel Description

```
1 <fsio>
2     <hs i="0x41200000" o="+0x8" width="1"/>
3     <fs i="0x41210000" o="+0x8" width="4"/>
4     <fs i="0x41220000" o="+0x8" width="4"/>
5     <fs i="0x41230000" o="+0x8" width="4"/>
6     <fs i="0x41240000" o="+0x8" width="4"/>
7 </fsio>
```

While `libfsio` iterates over the nodes of the XML[11] tree with the help of a common C++[12] library called `libpugixml` [44], it maps the addresses of each node and creates the communication channel as a composition of them in an object-oriented manner. Hence, a map is reflected by a C++ class named “fsio_map” while a channel is reflected by a class named “fsio”. An instance of latter class owns multiple instances of former class. While the AXI GPIO IP Core supports sizes from 1 to 32 signals represented by the CPU as 1 to 32 bits, the library distinguishes only between sizes of

[11] The Extensible Markup Language (XML) [45] encodes data human- and machine-readable.
[12] C++ [46] is a multi-paradigm general-purpose programming language.

the standard exact-width integer types of the C++ language, these are `std::uint8_t`, `std::uint16_t`, `std::uint32_t`, and `std::uint64_t` with 1, 2, 4, and 8 bytes in size, respectively. The last one is supported just in case future versions of the IP Core are extended to 64 bits. Accessing bits not mapped by the IP Core is safe, writing to them has no effect and reading from them results in zero values. Supporting only these sizes and not arbitrary byte sizes is due to the fact that the class "fsio_map" is actually a template class to be instantiated with one of these types as parameter. This has the advantage that its implementation template, as shown in algorithm 4.2, can make use of simple assignment and comparison operators of integer types instead of using variably sized assignment and comparison functions like `void* memcpy(void* dest, const void* src, std::size_t count)` and `int memcmp(const void* lhs, const void* rhs, std::size_t count)`, which would iterate over each single byte sacrificing performance for three reasons, function call overhead, managing a loop counter, and more important, unnecessarily more often triggering the underlying architecture of the virtual address translation.

Algorithm 4.2 Part of Map Implementation Template

```
64      auto size() const -> std::size_t override { return sizeof (T); }
65      void wave() override { *o = data = !data; }
66  #if FSIO_STAT
67      void feed() override { ++poll; *o = data = *i; }
68      auto back() -> bool override { ++poll; return data == *i; }
69  #else
70      void feed() override { *o = data = *i; }
71      auto back() -> bool override { return data == *i; }
72  #endif
73      auto stat() -> std::size_t {
74          auto poll = this->poll;
75          this->poll = 0;
76          return poll;
77      }
78      void own() override { data = *i; }
79      auto get(void* data) const -> std::size_t override {
80          *((T*)data) = this->data;
81          return sizeof (T);
82      }
83      auto put(const void* data) -> std::size_t override {
84          *o = this->data = *((const T*)data);
85          return sizeof (T);
86      }
```

This template of type parameter “T” implements the functionality of both handshake and data maps. The variables “i” and “o” are pointers to the virtual addresses of the input and output mappings. To access their value they are pointing to, they must be dereferenced by applying the “*” operator. The variable “data” is of type “T” and locally stores the value of the map. In case of a handshake map, “data” represents the value of the handshake signal, otherwise the value of a data map. The method `wave()` in line 65 does the handshake toggling by firstly toggling “data” locally before writing it to the mapped register. The methods `feed()` and `back()` in line 70 and 71 implement the feedback loop. Former feeds data back to the FPGA and latter compares written data with data fed back by the FPGA. Both are used by data maps while latter is additionally used by handshake maps to poll for an acknowledgement previously requested by the method `wave()`. Their alternative implementations in line 67 and 68, optionally selected by a compile-time switch `--enable-stat`, additionally count how often each method was called to give increased statistics for performance measurements. Finally the methods `auto get(void* data) const -> std::size_t` and `auto put(const void* data) -> std::size_t` in line 79 and 83 read from and write to the local storage “data” by taking a pointer to a data buffer as argument while returning the size of bytes they have accessed of it, which actually is the size of the local storage “data”.

To be able to compose a communication channel of differently sized maps, the class “fsio” defines an abstract interface “fsio::map”, shown in algorithm 4.3, which is implemented by the template class “fsio_map”, abstracting away its concrete type. In this way, the class “fsio” can transparently call methods of the class “fsio_map” without knowing about its concrete type, increasing the readability of the implementation of the proposed protocol.

Algorithm 4.3 Abstract Map Interface

```
50 class fsio {
51     class map {
52     public:
53         virtual ~map() {};
54         virtual auto size() const -> std::size_t = 0;
55         virtual void wave() = 0;
56         virtual void feed() = 0;
57         virtual auto back() -> bool = 0;
58         virtual auto stat() -> std::size_t = 0;
59         virtual void own() = 0;
60         virtual auto get(void* data) const -> std::size_t = 0;
61         virtual auto put(const void* data) -> std::size_t = 0;
62     };
```

4.2.3 Master Implementation

The class “fsio” reflects a channel of one handshake and one or more data maps, interfaced through the abstract class “fsio::map”. The implementation of the master write routine of the proposed protocol is shown from line 146 to 151 of algorithm 4.4. It firstly writes the data in chunks of its data map sizes by iterating over them and calling their write method through the abstract interface, `io->put(data)`. This method returns the size of the current map which is used to increment the data pointer. All data maps are stored in the vector “fs” whose single elements are accessed by its iterator “io”. Afterwards, the helper method `back()` is called, again iterating over all data maps to compare their feedback with their local data storage until equality.

Algorithm 4.4 Master Write (MW) Routine

```
121 void fsio::back() const {
122     for (bool back = false; !back; wait()) {
123         back = true;
124         for (auto& io: fs)
125             if (!io->back())
126                 back = false;
127     }
128     wave();
129 }
```

```
146 auto fsio::put(const void* data) const -> std::size_t {
147     for (auto& io: fs)
148         data = (const char*)data + io->put(data);
149     back_data ? back() : wave();
150     return fs_size;
151 }
```

Finally, another helper method `wave()`, shown in line 75 of algorithm 4.5, is called requesting the FPGA to read the data and waiting for its acknowledge. It firstly toggles the handshake by calling `hs->wave()`, being the request. Then it polls its feedback until equality by calling `hs->back()`, being the acknowledge. Each poll can be delayed by the method `wait()`. This delay is customizable. But if no delay is desired, the method call overhead would still result in an unnecessary delay. Thus, this method must be enabled by a compile-time switch `--enable-wait`. If not enabled, it is replaced by an ASM[13] instruction, as shown in line 71 of algorithm 4.5. Now the implementation of the method `wave()` is directly given in the C++ header file, allowing the compiler to substitute its content at its point of call instead of performing the method call [47]. Instead of an empty implementation, the ASM instruction is necessary, cause otherwise the for-loop in line 75 would have an empty body, that is having no instructions at all and since the `hs->back()` method has no other side effects in case previously mentioned statistics are not enabled by the compile-time

[13] An assembly (ASM) language corresponds to the instruction set of a particular CPU.

switch `--enable-stat`, the compiler would be allowed to optimize the for-loop away [48], fatally ignoring the handshake acknowledge.

Algorithm 4.5 Handshake Routine

```
68  #if FSIO_WAIT
69      void wait() const;
70  #else
71      void wait() const { __asm__ __volatile__ ("" ::: "memory"); };
72  #endif
73      void feed() const;
74      void back() const;
75      void wave() const { for (hs->wave(); !hs->back(); wait()); };
```

The implementation of the master read routine of the proposed protocol is shown from line 139 to 144 of algorithm 4.6. In contrast, in line 140 it firstly performs the handshake while alternately feeding read data back to the FPGA with the help of the method `feed()`, defined from line 112 to 119.

Algorithm 4.6 Master Read (MR) Routine

```
112 void fsio::feed() const {
113     auto feed = [this] () {
114         for (auto& io: fs)
115             io->feed();
116     };
117     for (hs->wave(), feed(); !hs->back(); wait())
118         feed();
119 }
```

```
139 auto fsio::get(void* data) const -> std::size_t {
140     feed_data ? feed() : wave();
141     for (auto& io: fs)
142         data = (char*)data + io->get(data);
143     return fs_size;
144 }
```

This method defines another helper function iterating over all data maps and calling their feed method, `io->feed()`. After the handshake routine is done, that means

it requested the FPGA to write data and waited for its acknowledgment of completion, it finally reads the data in chunks of its data map sizes by iterating over them and calling their read method, `io->get(data)`. This method returns the size of the current map which is used to increment the data pointer.

4.2.4 Slave Implementation

The slave implementation of the protocol for the FPGA is written in a hardware description language (HDL) called Very High Speed Integrated Circuit Hardware Description Language (VHSIC HDL), or just VHDL [49]. The top level VHDL file can access the mapped registers by instantiating the block design. Each map provides its own pair of input and output signal vectors. Thus, the handshake map is represented by two vectors of length 1 while a data map is represented by two vectors of length 32. The vectors of the four data maps can be merged to two vectors of length $4 \cdot 32 = 128$, one merging the input and the other merging the output vectors. Hence, the communication channel is represented by two handshake vectors of length 1, or just the two signals themselves, “hsi” and “hso”, and two vectors of length 128, “fsi” and “fso”.

To read from a channel, its vectors are connected to the entity “fsio_get” implementing the slave read routine, shown in algorithm 4.7. The generic constants “cap” and “len” must be adjusted for a given channel. While “cap” is the width of the channel, “len” might be chosen smaller in case not all signals of the channel are used. The data can be read from the vector “dat” of length “len”. When data is available, the signal “req” is set. This request can be acknowledged when the vector “dat” has been read by setting the signal “ack” for exactly one clock cycle. It causes the signal “req” to be reset. The clock signal “clk” must be connected to the AXI clock of the block design.

Algorithm 4.7 Slave Read (SR) Routine

```
23 entity fsio_get is
24     generic (
25         cap: integer := CAP;
26         len: integer := LEN
27     );
28     port (
29         clk: in std_logic;
30         hsi: in std_logic;
31         hso: out std_logic;
32         fsi: in std_logic_vector(cap - 1 downto 0);
33         fso: out std_logic_vector(cap - 1 downto 0);
34         dat: out std_logic_vector(len - 1 downto 0);
35         req: out std_logic;
36         ack: in std_logic
37     );
38 end fsio_get;
39
40 architecture behavioral of fsio_get is
41 begin
42     dat <= fso(len - 1 downto 0);
43     req <= hso xor hsi;
44     fso <= fsi;
45     ctl: process(clk)
46     begin
47         if rising_edge(clk) then
48             hso <= hso xor (req and ack);
49         end if;
50     end process ctl;
51 end behavioral;
```

The data written by the CPU is continuously fed back in line 44. A request is detected in line 43 by a combinational logic of an Exclusive OR (XOR) gate[14] with “hsi” and “hso” as inputs and “req” as output. Its truth table is shown in table 4.1. When the CPU toggles the handshake signal, “hsi” changes compared with “hso”. Thus, a request is given for case 2 and 3 of the truth table setting “req”.

[14] A logic gate implements a logical operation on one or more inputs, producing a single output.

Case	A	B	A XOR B
1	0	0	0
2	0	1	1
3	1	0	1
4	1	1	0

Table 4.1: Exclusive OR (XOR) Gate

Another XOR gate is used in line 48 to perform the acknowledgment. This XOR gate functions as conditional inverter. The data input of a D flip-flop[15] representing the signal “hso” is connected to the output of the XOR gate while the output of the flip-flop is connected to one of the two inputs of the gate. The other input is used as inversion enabler in case `(req and ack)` is logical “1”. Let input “A” be the signal to be inverted and input “B” the inversion enabler. When “B” is not set, that is case 1 and 3, the output is of the same value of “A”. But when “B” is set, that is case 2 and 4, the output is the inversion of “A”. Thus, setting the signal “ack” within a request, that is `(hso /= hsi)`, causes the signal “hso” to be inverted, following the previous toggle of “hsi” done by the CPU.

To write to a channel, its vectors are connected to the entity “fsio_put” implementing the slave write routine, shown in algorithm 4.8. It uses equal combinational logics of XOR gates for request detections and acknowledgements. But instead of feeding “fsi” back to “fso”, it compares them and delays the acknowledgment until they match. Thus, in contrast to “fsio_get”, “ack” must be kept set until “req” is reset.

[15] The D flip-flop captures the value of its input, becoming its output at the next clock cycle.

Algorithm 4.8 Slave Write (SW) Routine

```
23 entity fsio_put is
24     generic (
25         cap: integer := CAP;
26         len: integer := LEN
27     );
28     port (
29         clk: in std_logic;
30         hsi: in std_logic;
31         hso: out std_logic;
32         fsi: in std_logic_vector(cap - 1 downto 0);
33         fso: out std_logic_vector(cap - 1 downto 0);
34         dat: in std_logic_vector(len - 1 downto 0);
35         req: out std_logic;
36         ack: in std_logic
37     );
38 end fsio_put;
39
40 architecture behavioral of fsio_put is
41 begin
42     fso(len - 1 downto 0) <= dat;
43     req <= hso xor hsi;
44     ctl: process(clk)
45     begin
46         if rising_edge(clk) then
47             if fso = fsi then
48                 hso <= hso xor (req and ack);
49             end if;
50         end if;
51     end process ctl;
52 end behavioral;
```

4.2.5 File Transfer Application

The application `fsio` makes use of the library `libfsio` to transfer files between CPU and FPGA. As shown in figure 4.8, it accepts a channel description file as command line argument labeled “FSIO” and transfers data either from the channel to a ”FILE” or vice versa, whether “MODE” is “get” or “put”, respectively. Optionally, it can dump transfer statistics like the transfer rate for performance measurements to a file labeled “DUMP”. The transfer rate, the total number of bytes already transferred, the

elapsed time, the progress in percentage, and the estimated time of arrival (ETA) are calculated and displayed either every time the buffer will be emptied or at the end of the transfer only. Latter for not distorting performance measurements due to increased CPU usage for the calculations and terminal refreshing. In “get” mode, the percentage and ETA is only displayed if the so-called end of file (EOF), required for these calculations, is given by option `-e, --eof`. In “put” mode instead, the size of “FILE” is used as default for “EOF” in case it is determinable, which is the case for regular files. The buffer size can be fine-tuned in multiples of the channel width by option `-b, --buf`. To cancel the current transfer, pressing `CTRL+C` safely terminates the application in case it is not busy with polling. Otherwise, double-pressing `CTRL+C` immediately but unsafely terminates the application. Unsafe, in the sense of possibly leaving the FPGA in an uncompleted handshake and not cleaning up all resources of the application resulting in memory leaks.

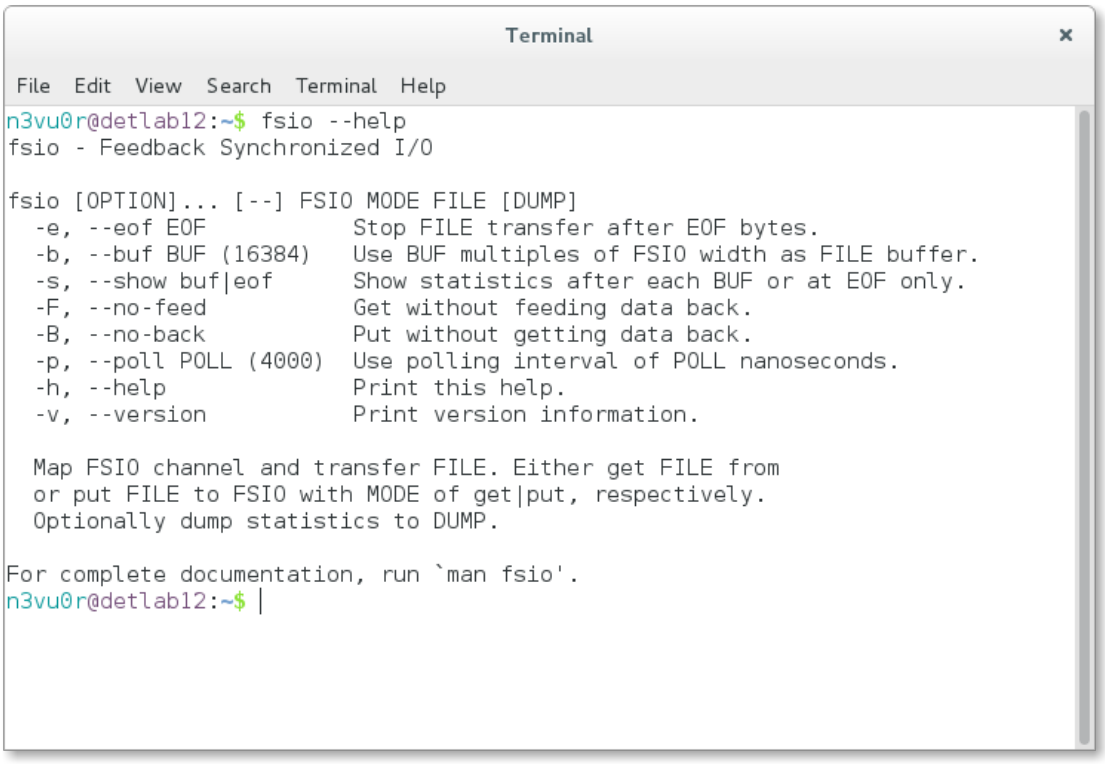


Figure 4.8: Usage of File Transfer Application

For quick a generation of test files, especially on embedded devices, another application named `fsio-tvgen` has been written. As shown in figure 4.9, it accepts three command line arguments labeled “ROWS”, “COLS”, and “SIZE”. With these it is possible to generate a binary file of incremental “SIZE”-byte sized integer values. All integers of a column have the value of its row number. In this way, they can be

aligned to the channel width by choosing “COLS” appropriately to facilitate debugging by distinguishing dumped values of different channels. With “COLS” of 1, no values are repeated. The argument “ROWS” adjusts the final file size.

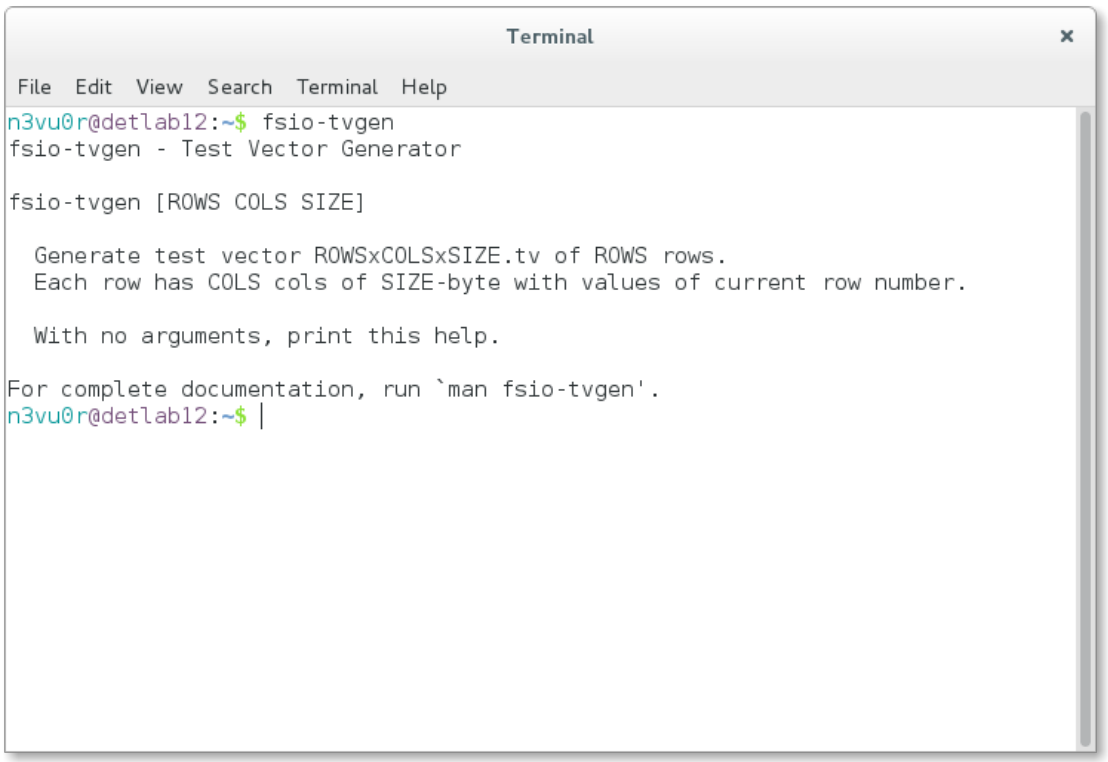


Figure 4.9: Usage of Test Vector Generator

A test vector, aligned to a channel of width $4 \cdot 4\,\mathrm{B} = 16\,\mathrm{B}$, would be of four columns of 4-byte sized integers. With one million rows, its final size sums up to $16\,\mathrm{MB}$. A hexadecimal dump of the first sixteen rows of this vector is shown in figure 4.10. The left column is the byte offset of a row. The other four columns are the 4-byte sized integer values represented by eight hexadecimals. They are incremented each row, starting with zero.

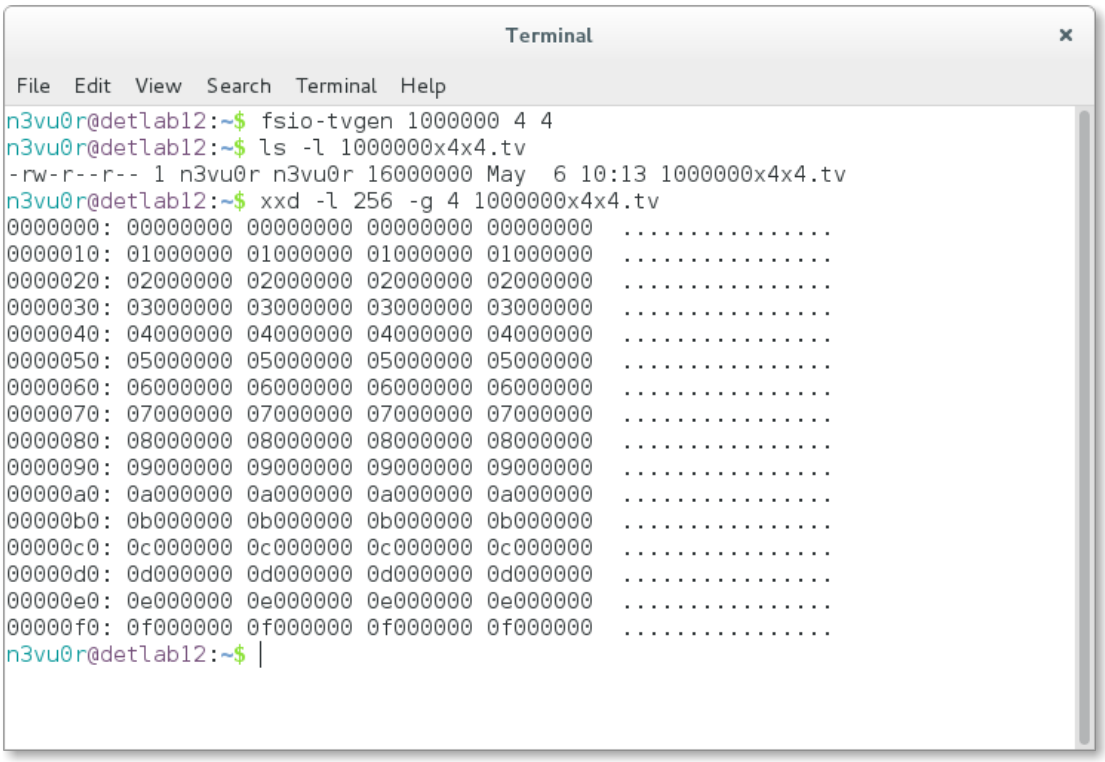


Figure 4.10: Example Test Vector

4.3 Clock Generation

Clock generation is done by a highly-flexible quad clock generator from Silicon Labs called Si5338 [50, 51, 52], entirely configurable through an I²C interface. It is capable of locking to an external input clock for generating up to four synchronous output clocks of user-programmable frequencies. This enables generation of independent clocks for the MGT receivers connected to the ECAL and HCAL through the optical plant, the MGT transmitters connected to the L1Topo, the MGT transmitters interfacing the ROD daughter module, and for a pass-through of the global input clock to which all these outputs are synchronous.

At power-up the device copies its entire configuration from a built-in non-volatile memory (NVM) into its random-access memory (RAM). The NVM is an one-time programmable memory (OTP) serving as an user-definable default configuration storage. Once programmed by a so-called field programmer, it can never be reprogrammed again. After power-up the device is still configurable by manipulating registers of the RAM via the I²C interface. Silicon Labs provides a software tool with a graphical user interface (GUI) called ClockBuilder Desktop Software [53]

to simplify the device configuration. It is capable of connecting and writing device configurations to the Si5338 evaluation board [54]. This board from Silicon Labs provides connectivity for all input and output clock signals in order to test several features. Alternatively, the entire configuration can be stored in either a register map file or a C[16] code header file. While former is used by the tool itself and further distinguished to either just load a previously saved device configuration or to program the NVM with the field programmer [55] from Silicon Labs. Latter is intended for inclusion in an own C application.

For automated control of clock generation at run time of the jFEX, the software package `si53xx` has been developed. The name was chosen to emphasize potential compatibility to similar devices like the Si5356 [56] from Silicon Labs. This package consists of three tools, `si53xx-map` for converting register maps, `si53xx-cmp` for generating transition maps, and `si53xx` for actually controlling the device.

4.3.1 Register Map Creation

The ClockBuilder Desktop Software from Silicon Labs is divided into six tabs labeled “Frequency Plan”, “Output Drivers”, “Power”, “Inc and Dec”, “Spread Spectrum”, and “Status”. The first tab is used to define the input clock source and how output clocks are derived from it. The customizations done on this tab for the default operation of the jFEX are shown in figure 4.11. It expects the global input clock of $40.0787\,\mathrm{MHz}$ on pins “IN1” and “IN2” and generates four output clocks labeled “CLK0”, “CLK1”, “CLK2”, and “CLK3” of $280.5509\,\mathrm{MHz}$, $160.3148\,\mathrm{MHz}$, $240.4722\,\mathrm{MHz}$, and $40.0787\,\mathrm{MHz}$ by assigning integral multipliers of 7, 4, 6, and 1, respectively. The first three clocks are used by the processor FPGAs as MGT reference clocks to generate the final transfer rate clocks of $11.2\,\mathrm{GHz}$, $12.8\,\mathrm{GHz}$, and $9.6\,\mathrm{GHz}$ with multipliers of 40, 80, and 40, respectively.

[16] C [57] is a general-purpose programming language.

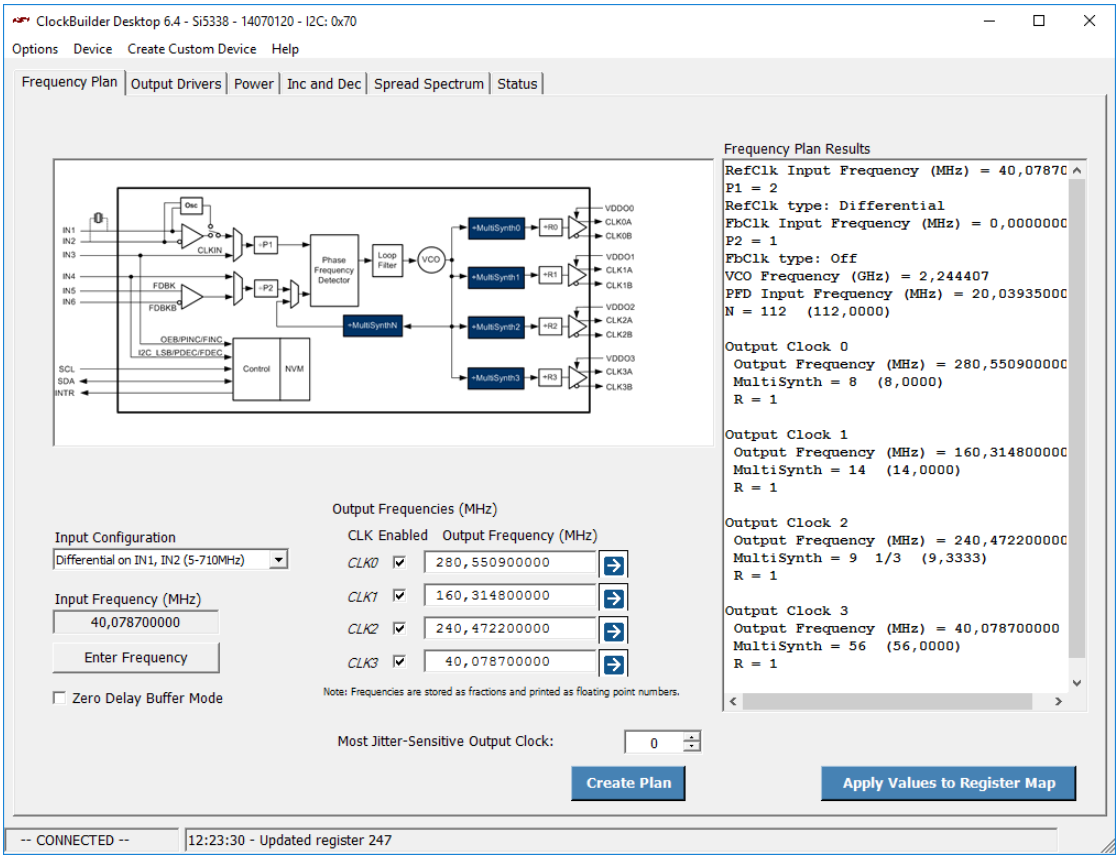


Figure 4.11: Frequency Plan [53]

As shown in figure 4.12, the output drivers can be configured as well. For the MGT reference clocks, the current mode logic (CML) at $2.5\,\mathrm{V}$ was chosen as output type. CML is a differential digital logic family widely used in high-speed systems like serial data transceivers or frequency synthesizers. The pass-through of the global clock is distributed by low-voltage differential signaling (LVDS) at $2.5\,\mathrm{V}$.

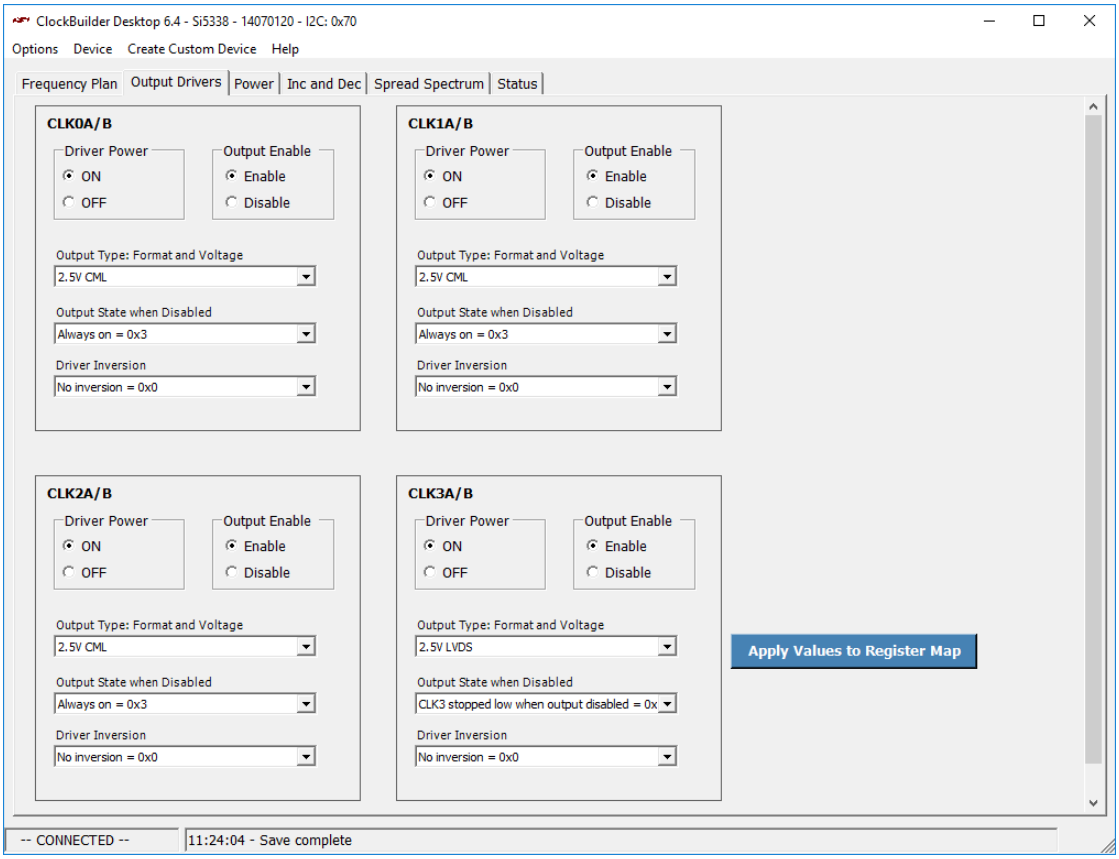


Figure 4.12: Output Drivers [53]

The Si5338 device can be powered and interfaced at different voltages, see figure 4.13. The jFEX supplies it with $2.5\,\mathrm{V}$ but the mezzanine card interfaces it at $3.3\,\mathrm{V}$.

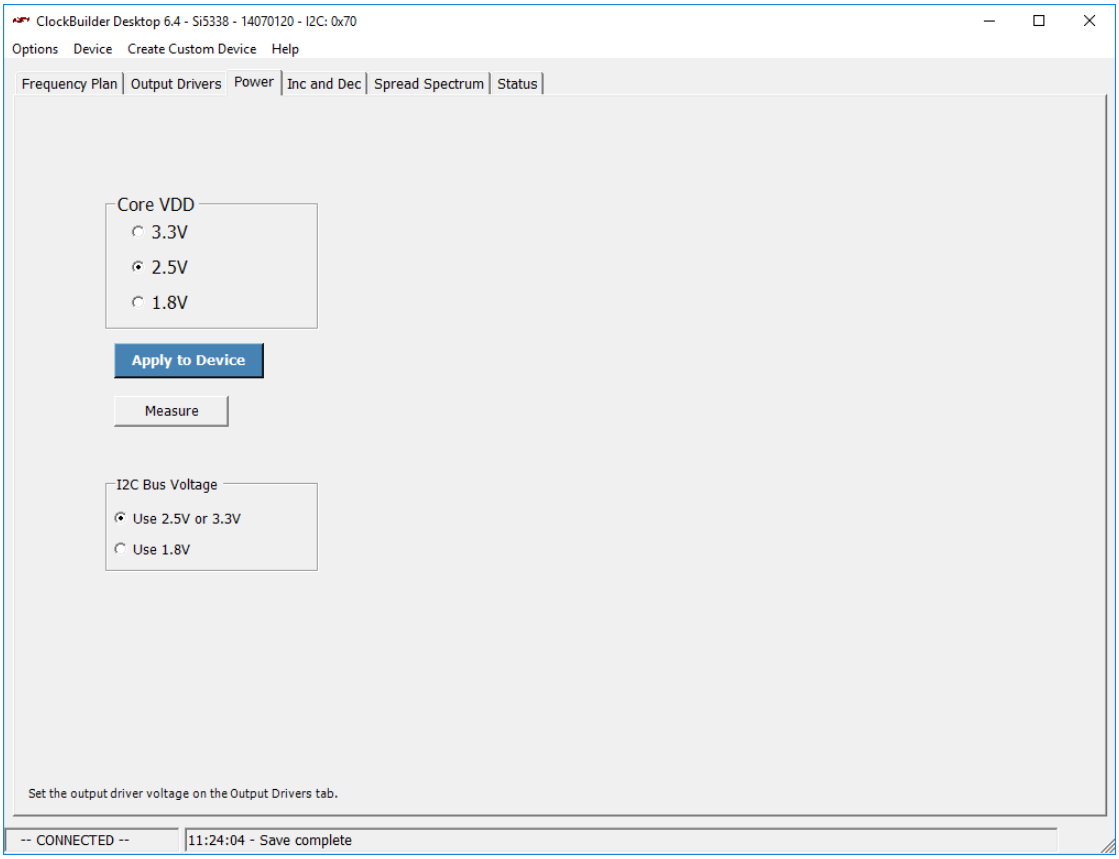


Figure 4.13: Power [53]

The “Inc and Dec” tab in figure 4.14 enables independent configuration of an initial phase offset and phase walks for each clock output. These features are not used by the default configuration of the jFEX.

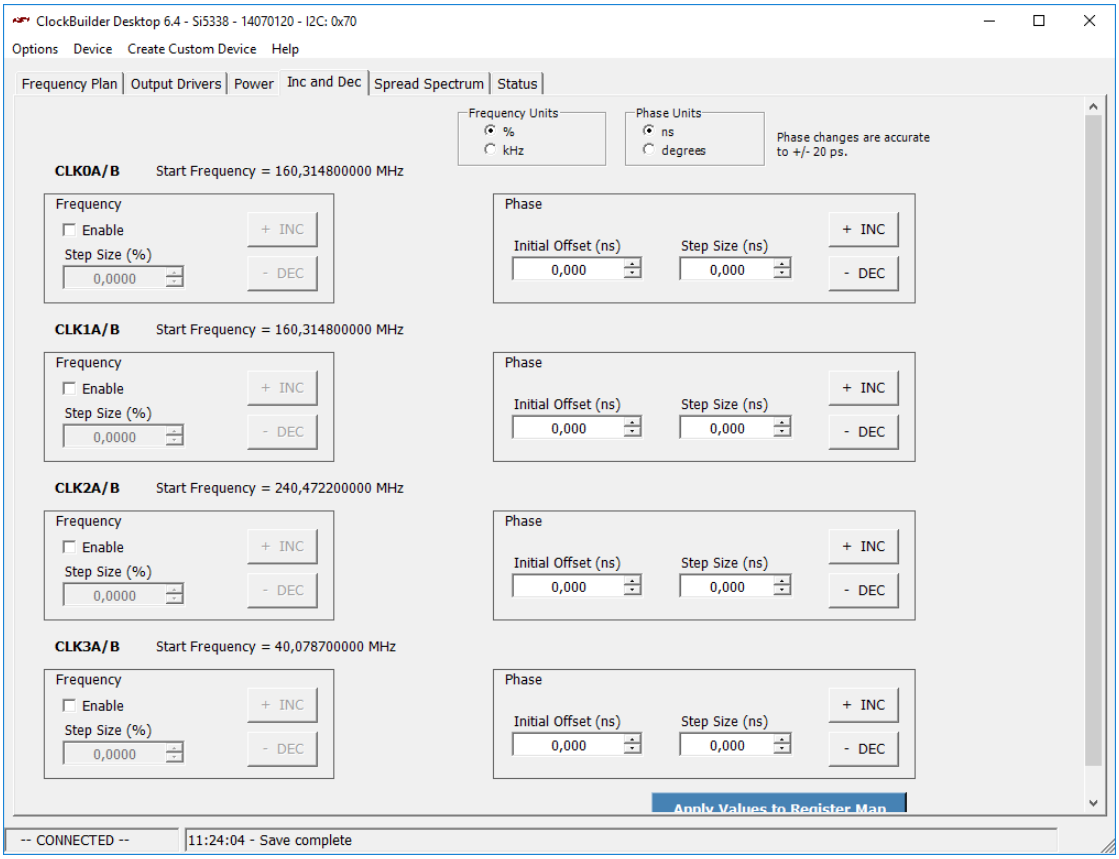


Figure 4.14: Inc and Dec [53]

In order to reduce electromagnetic interference (EMI) caused by higher-order harmonics of clock signals, a spread spectrum clock can be generated by dithering its frequency between defined margins, as shown in figure 4.15. Further parameters like modulation rate or spread profile can be tuned as well by analyzing possible EMI of the final hardware. Thus, the default configuration of the jFEX keeps this feature disabled.

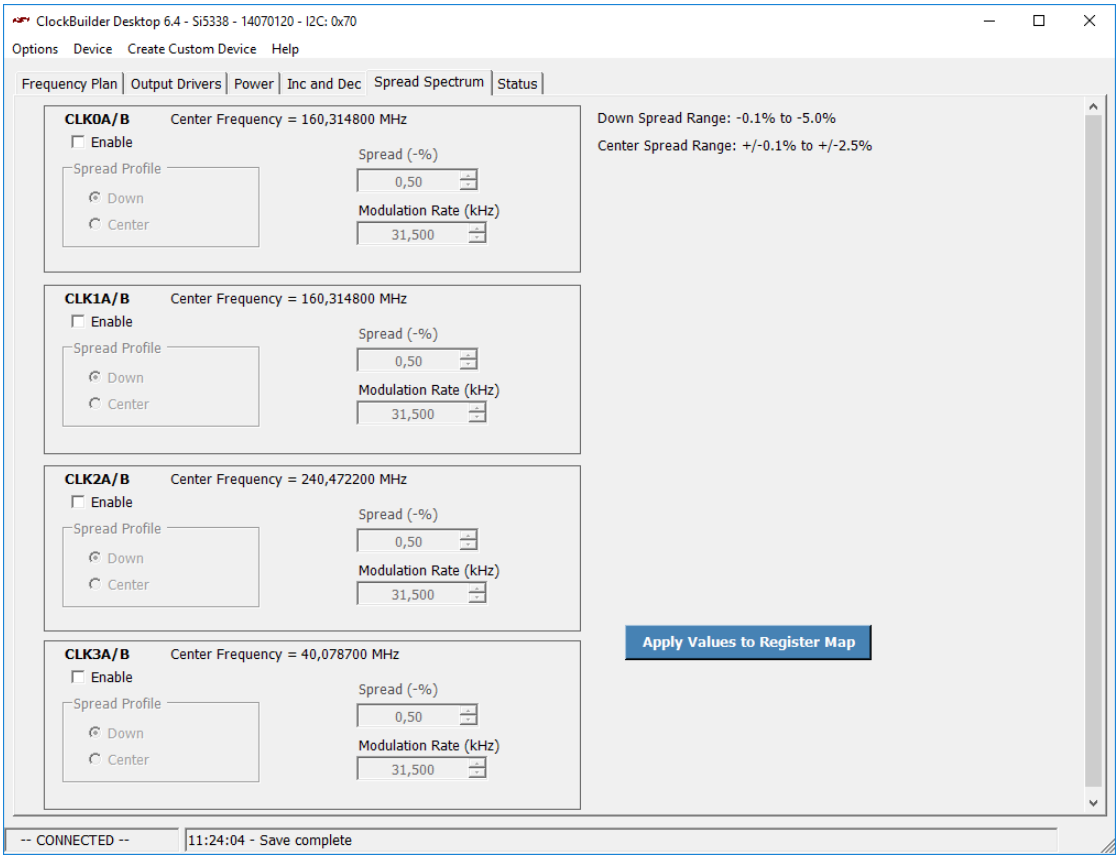


Figure 4.15: Spread Spectrum [53]

The last tab, shown in figure 4.16, displays the device status made up of four flags. The flags “LOS_CLKIN” and “LOS_FDBK” indicate loss of signal (LOS) of the input clock (CLKIN) and of the feedback (FDBK), respectively. The flag “PLL_LOL” indicates loss of lock (LOL) of the PLL[17] and the flag “SYS_CAL” indicates that the system (SYS) is calibrating (CAL), that is the PLL is in the process of acquiring its lock. In addition, these signals can trigger an interrupt on a dedicated pin of the device. In case a certain status flag is expected to indicate an error, like “LOS_FDBK” if the feedback is not used at all, it can safely be ignored by selecting it in the interrupt mask. To not miss an error in case it resolved itself, the sticky status is only set automatically but must be reset manually by clicking the corresponding signal in the GUI.

[17] A phase-locked loop (PLL) controls the phase of an output signal related to an input signal.

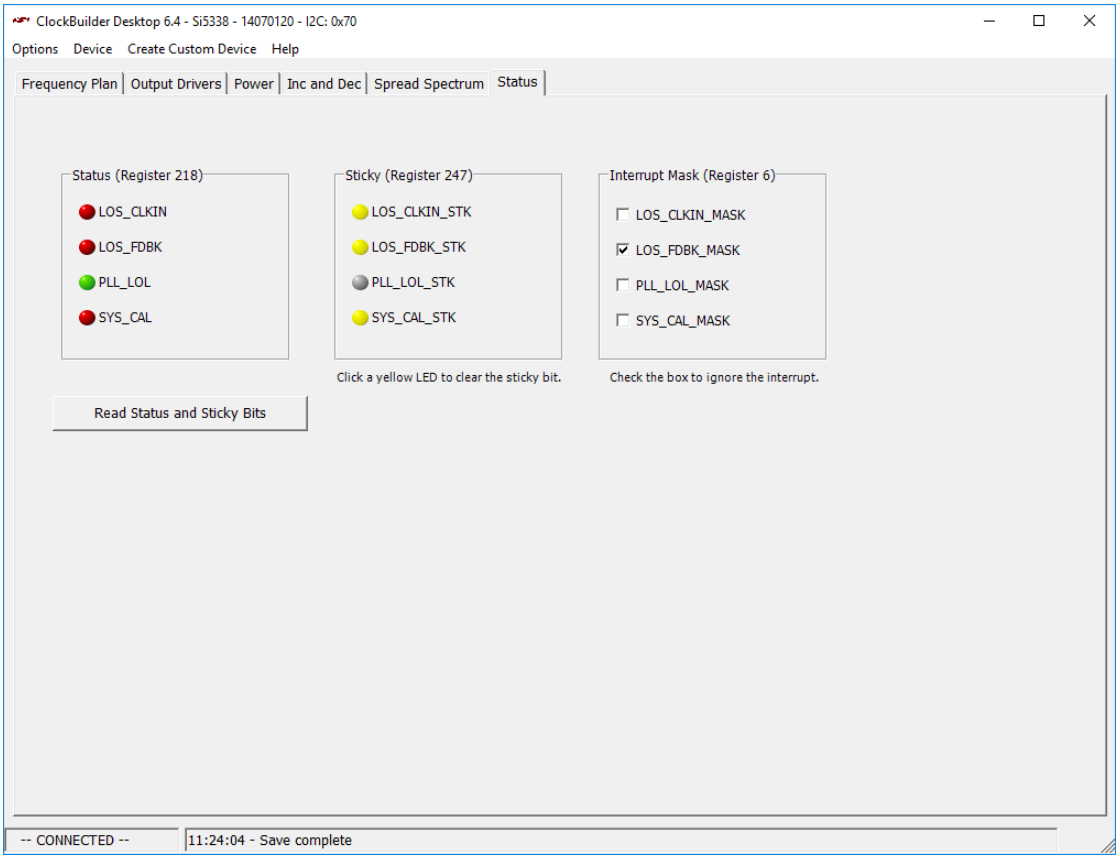


Figure 4.16: Status [53]

Finally, after configuration is done, it can be saved by clicking the menu entry “Options” followed by “Save register map file (not for factory programming)...”, as shown in figure 4.17. In this way a previously saved configuration can be restored by clicking “Open register map file...”. To create a register map file for programming the NVM, “Save registers for factory programming...” is chosen instead. This will additionally offer a wizard to request a custom part number for ordering preconfigured devices. For inclusion in an own C application, “Save C Code Header File...” is used.

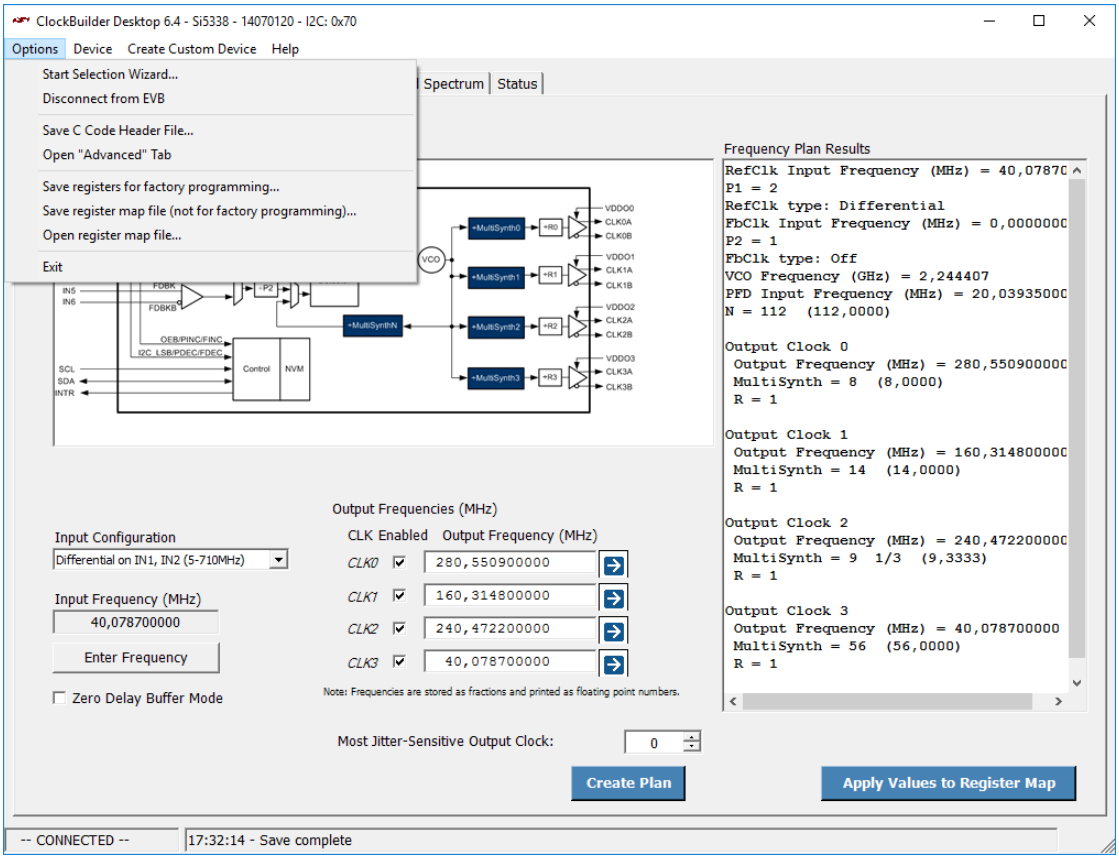


Figure 4.17: Options [53]

4.3.2 Register Map Conversion

The C code header file is meant to be included in an own C code application. This is useful for static configurations only, since updating the configuration would require recompiling the application. Another approach is to write an application capable of parsing a configuration file at run time. To keep the source code for parsing a configuration file simple and independent from the current format, a conversion script named `si53xx-map` has been written. The C code header file stores the information, necessary to write a single register, in a C structure of three integer values of one byte each, as shown from line 84 to 88 of algorithm 4.9. Starting from line 90, an array is defined storing the information of all the 350 registers of a certain configuration. For each register, its address in RAM, its actual value, and an 8-bit mask is specified from left to right in the order mentioned. While the address is given in decimals ranging from 0 to 255, the other two entries are given in hexadecimals ranging from `0x00` to `0xFF`. Since the smallest transferable unit of the I²C bus is a byte made up of 8 bits, a mask is required to mark the bits actually intended to be

written and to ignore the others. To access a register above the 1-byte sized integer range from 0 to 255, the page register at address 255 must be set to `0x01`, causing the device to internally add an offset of 255 to the register addresses of subsequent I²C commands. To access again lower addressed registers, the page register must be reset to `0x00`.

Algorithm 4.9 Part of C Code Header File

```
82  #define NUM_REGS_MAX 350
83
84  typedef struct Reg_Data{
85     unsigned char Reg_Addr;
86     unsigned char Reg_Val;
87     unsigned char Reg_Mask;
88  } Reg_Data;
89
90  Reg_Data const code Reg_Store[NUM_REGS_MAX] = {
91  {  0,0x01,0x00},
92  {  1,0x00,0x00},
93  {  2,0x26,0x00},
94  {  3,0x70,0x00},
95  {  4,0x00,0x00},
96  {  5,0x00,0x00},
97  {  6,0x08,0x1D},
98  {  7,0x20,0x00},
99  {  8,0x10,0x00},
100 {  9,0xD0,0x00},
101 { 10,0x0C,0x00},
```

In contrast, the converted register map file stores the three entries of a register in a line of white space separated hexadecimals, as shown in algorithm 4.10. The order is unchanged but the address entry is given in hexadecimals, too. Having all entries simply white space separated makes further parsing almost trivial. But more important, having a strict format makes register maps reproducibly comparable by common tools like `diff`, as used in section 4.3.3. Additionally, all lines with a mask of `0x00` are dropped since writing no bits of a byte at all has no effect anyway.

Algorithm 4.10 Part of Converted Register Map File

```
1  06 08 1d
2  1b 70 80
3  1c 03 ff
4  1d 41 ff
5  1e b0 ff
6  1f c0 ff
7  20 c0 ff
8  21 c0 ff
9  22 c0 ff
10 23 55 ff
```

The script accepts multiple header files as arguments and converts them effectively in-place, renaming their file extensions from ".h" to ".map", as shown in figure 4.18.

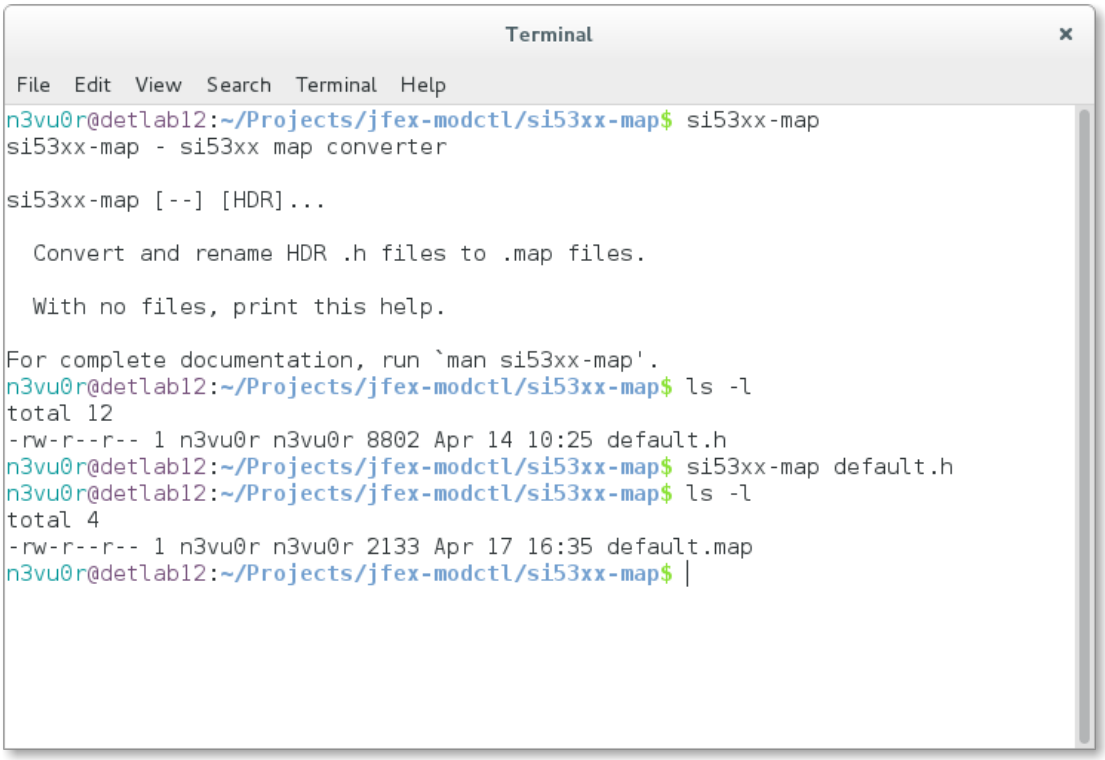


Figure 4.18: Usage of Register Map Converter

The basic steps performed to do the conversion are shown in algorithm 4.11. From line 33 to 37 four functions are defined. First two process file names while last two process file contents. `hdr_file()` filters file names by their extension, passing

only those ending with ".h". `map_file()` replaces the extension of a file name from ".h" to ".map". `hdr_data()` extracts the register entries of a C code header file line by line. It invokes a stream editor (`sed`) [58], capable of evaluating so-called regular expressions[18], with four semicolon separated commands. The first command, `'/{[^}]\+}/!d'`, deletes all lines not enclosed by the curly braces "{" and "}". The second command, `'s/.*{\([^}]\+\)}.*/\1/'`, extracts the three comma separated entries within these curly braces. The third command, `'s/,/ /g'`, replaces each comma with a white space. And the fourth command, `'/00 *$/d'`, deletes all lines with a mask of `0x00`. This is almost the desired result, all three entries of a line are white space separated. But some entries like the register addresses are indented by additional white spaces to align them to the right, which is common for columns of decimals to improve readability. These additional white spaces are removed by the last of the four functions, `map_data()`, which also formats all entries as lowercase hexadecimals without the `0x` prefix. Now, having the core functionality implemented, it can be applied to each file by looping over their file names, given as command line arguments, starting from line 39 of algorithm 4.11. If a file name ends with ".h", checked in line 40, its content is extracted, reformatted and saved to a new ".map" file by line 41 and 42, respectively. Finally, the old ".h" file is deleted by line 43.

Algorithm 4.11 Part of Register Map Converter Script

```
33 hdr_file() { echo "$1" | grep -qG '\.h$'; }
34 map_file() { echo "$1" | sed 's/\.h$/.map/'; }
35
36 hdr_data() { sed '/{[^}]\+}/!d;s/.*{\([^}]\+\)}.*/\1/;s/,/ /g;/00 *$/d' "$1"; }
37 map_data() { echo "$1" | xargs -r printf '%02x %02x %02x\n'; }
38
39 for hdr_file; do
40     if hdr_file "$hdr_file"; then
41         hdr_data=$(hdr_data "$hdr_file") || exit 64
42         map_data "$hdr_data" > "$(map_file "$hdr_file")" || exit 65
43         rm "$hdr_file" || exit 66;
44     fi
45 done
```

[18] A regular expression is a sequence of characters defining a search pattern.

4.3.3 Transition Map Generation

The converted register maps of previous section 4.3.2 contain an entire device configuration. This means, for every single change of the configuration to be made at run time, an entire new register map must by saved and written to the device. This has several drawbacks. In addition to the changed registers, all the unchanged registers are written as well, unnecessarily increasing the time of blocking a whole chain of I²C slave devices. Multiple register maps would differ only slightly for a certain feature configured differently. But changing a common feature of them, would require to recreate all the different register maps each time. To avoid this, another script named `si53xx-cmp` has been written for further processing of converted register maps. This script expects a register map as command line argument labeled "OLD" and one or more additional register maps as command line arguments labeled "NEW...", as shown in figure 4.19.

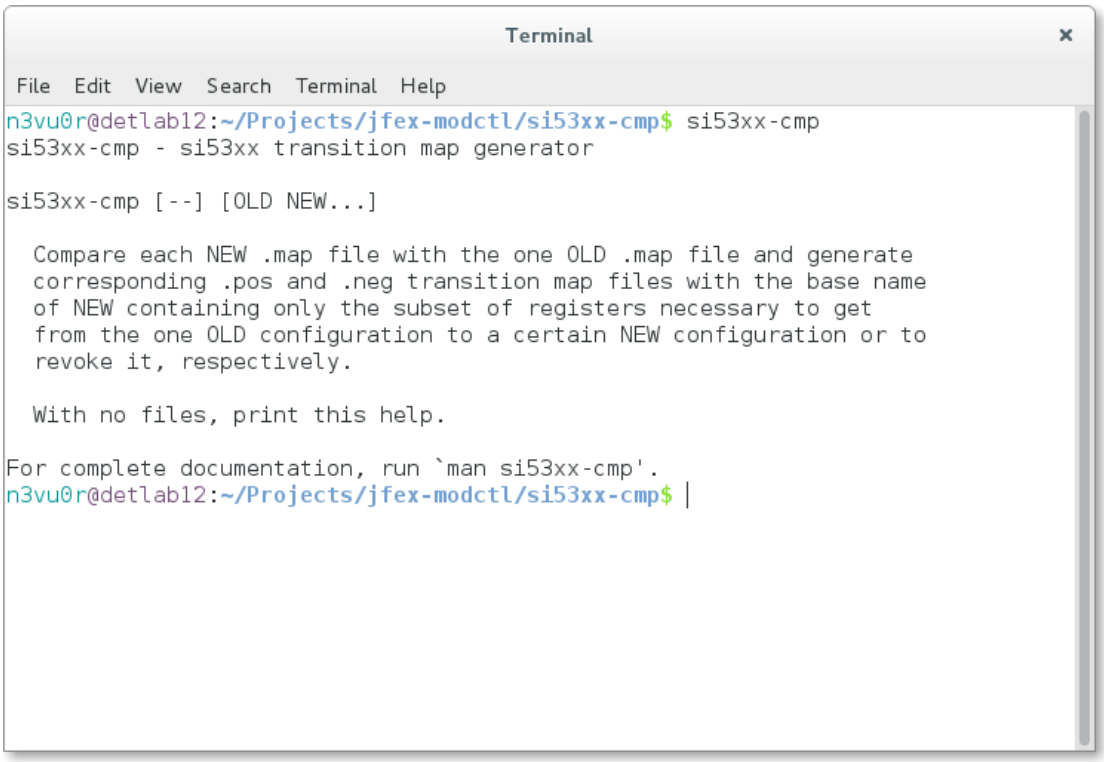


Figure 4.19: Usage of Transition Map Generator

It compares each "NEW" map with the one "OLD" map and generates corresponding positive ".pos" and negative ".neg" transition map files with the base name of "NEW" containing only the subset of registers necessary to get from the one "OLD" configuration to a certain "NEW" configuration or to revoke it, respectively. An example is

given in figure 4.20, where “default.map” is labeled “OLD” while “zero-delay.map” and “clk0@160.map” are labeled “NEW...”. When comparing the file sizes in the fifth column from left of the resulting transition map pairs, $36\,\mathrm{B}$ and $99\,\mathrm{B}$, with the file sizes of their original register maps, both of $2133\,\mathrm{B}$, file size reductions by factors of 59.25 and 22.54 for the features “zero-delay” and “clk0@160” are observed, respectively. As expected, this drastically reduces the number of registers necessary to be written.

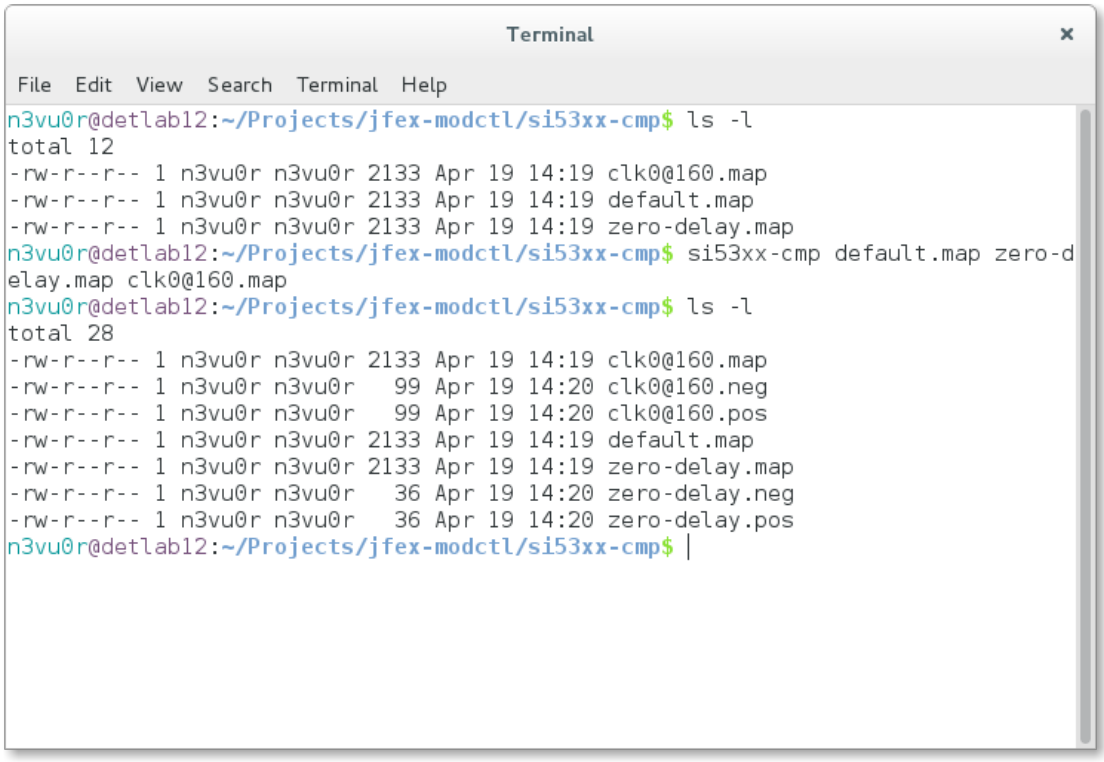


Figure 4.20: Example Transition Maps

In this way, the feature or change of parameters itself, not the entire new configuration, is represented by its own pair of “.pos” and “.neg” files, former for applying and latter for revoking it, as visualized in figure 4.21.

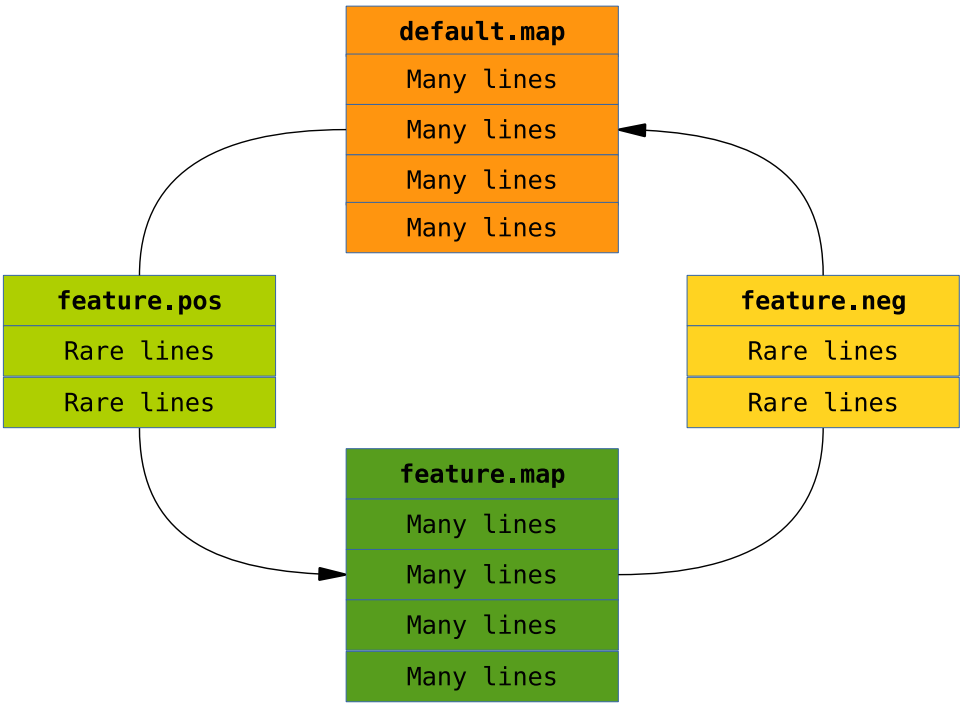


Figure 4.21: Transition Map Pair

Since a read-modify-write procedure against a mask for each register to be written is performed just in case its mask is not `0xFF` [52], several ".pos" or ".neg" files might share common registers with different values without interference, as long as the features they describe are bitwise orthogonal. In contrast, overlapping features simply must be revoked by invoking their ".neg" file before invoking the other's ".pos" file. Effectively, this script automates identifying and managing registers of all possible features by representing them as transition map files without studying all the 350 registers described in the reference manual [51] of the device and without manually implementing a single feature at all.

As shown in algorithm 4.13, the script firstly defines eight functions from line 35 to 46. `map_file()` filters file names by their file extension, passing only those ending with ".map" while `pos_file()` and `neg_file()` replace them from ".map" to ".pos" and ".neg", respectively. The last five functions operate on file contents. Since register addresses are spread over two pages as explained in section 4.3.2, the page must be selected appropriately before its registers can be accessed. As shown in algorithm 4.12, register maps require page `0x00` to be already selected, then they always select

page 0x01 even if they do not contain a register of that page, and finally they select page 0x00 again to meet the requirement of a possible subsequent register map.

Algorithm 4.12 Register Pages

```
06 08 1d
1b 70 80
.. .. ..
d9 00 ff
f2 00 02
ff 01 ff # Select page 01.
1f 00 ff
20 00 ff
.. .. ..
5a 00 ff
5b 00 0f
ff 00 ff # Select page 00.
```

Since all register maps always have these two lines selecting a page, they would be dropped when extracting changes only. Thus, when reading the one register map labeled "OLD" by the function `old_data()`, these two lines are explicitly dropped by deleting lines beginning with the page register address of `0xFF`, resulting in a forced change when compared with each of the register maps labeled "NEW...", done by the function `new_data()`. This function actually invokes the common tool `diff` [36]. It accepts two files as command line arguments and compares them line by line. It reports all non-matching lines along with their line numbers. Lines being present in the right but not in the left file are preceded by a right arrow `'> '` while lines being present in the left but not in the right file are preceded by a left arrow `'< '`. The last two functions, `pos_data()` and `neg_data()`, filter these two kinds of lines, respectively. Here, to filter means to pass matches while to filter out means to drop matches. Additionally, `neg_data()` passes lines of the opposite kind if beginning with `'ff'`. After filtering, both functions remove the prefix of each line. In case no registers of page `0x01` are written at all, the two lines selecting the pages are needless and can be deleted since they enclose no other lines. This is done by the function `clr_page()` passing the command `'/^ff/{N;/\nff/d}'` to the already mentioned tool `sed` looking for lines beginning with `'ff'`. On match, the semicolon separated subcommands enclosed in curly braces, `'N'` and `'/\nff/d'`, are invoked. Former appends the next line to the pattern space allowing the latter subcommand to operate on both lines. This is necessary since by default, `sed` operates on single

lines only. Latter subcommand looks for a sequence of a newline character `'\n'` followed by `'ff'`. On match, these two lines are deleted. After having comparison and filtering implemented, it can be applied to each register map given as command line arguments. From line 48 to 53 of algorithm 4.13, the one register map labeled "OLD" is processed. From line 55 to 62 the additional register maps labeled "NEW..." are processed by looping over their file names. Line 57 filters out file names not ending with ".map" and ensures to not list the one "OLD" file additionally as a "NEW" file when using the wild card character "*" as "NEW..." label, selecting all files of a directory. Line 58 does the actual comparison followed by the positive and negative filtering in line 59 and 60, whose results are redirected to the corresponding ".pos" and ".neg" files, respectively.

Algorithm 4.13 Part of Transition Map Generator Script

```
35 map_file() { echo "$1" | grep -qG '\.map$'; }
36
37 pos_file() { echo "$1" | sed 's/\.map$/.pos/'; }
38 neg_file() { echo "$1" | sed 's/\.map$/.neg/'; }
39
40 old_data() { sed '/^ff/d' "$1"; }
41 new_data() { echo "$1" | diff - "$2"; test $? -ne 2; }
42
43 clr_page() { sed '/^ff/{N;/\nff/d}'; }
44
45 pos_data() { echo "$1" | sed '/^>/!d;s/^> //' | clr_page; }
46 neg_data() { echo "$1" | sed '/^\(<\|> ff\)/!d;s/^. //' | clr_page; }
47
48 old_file=$1
49 map_file "$old_file" || {
50     >&2 echo 'OLD is not a .map file'
51     exit 64
52 }
53 old_data=$(old_data "$old_file") || exit 65
54
55 shift
56 for new_file; do
57     if map_file "$new_file" && test "$old_file" != "$new_file"; then
58         new_data=$(new_data "$old_data" "$new_file") || exit 66
59         pos_data "$new_data" > "$(pos_file "$new_file")" || exit 67
60         neg_data "$new_data" > "$(neg_file "$new_file")" || exit 68
61     fi
62 done
```

4.3.4 Device Control

To finally control the device, the C++ application `si53xx` has been written. While the user interface, shown in figure 4.22, is implemented in this binary, the file parsing and communication routines are implemented in the library `libsi53xx`.

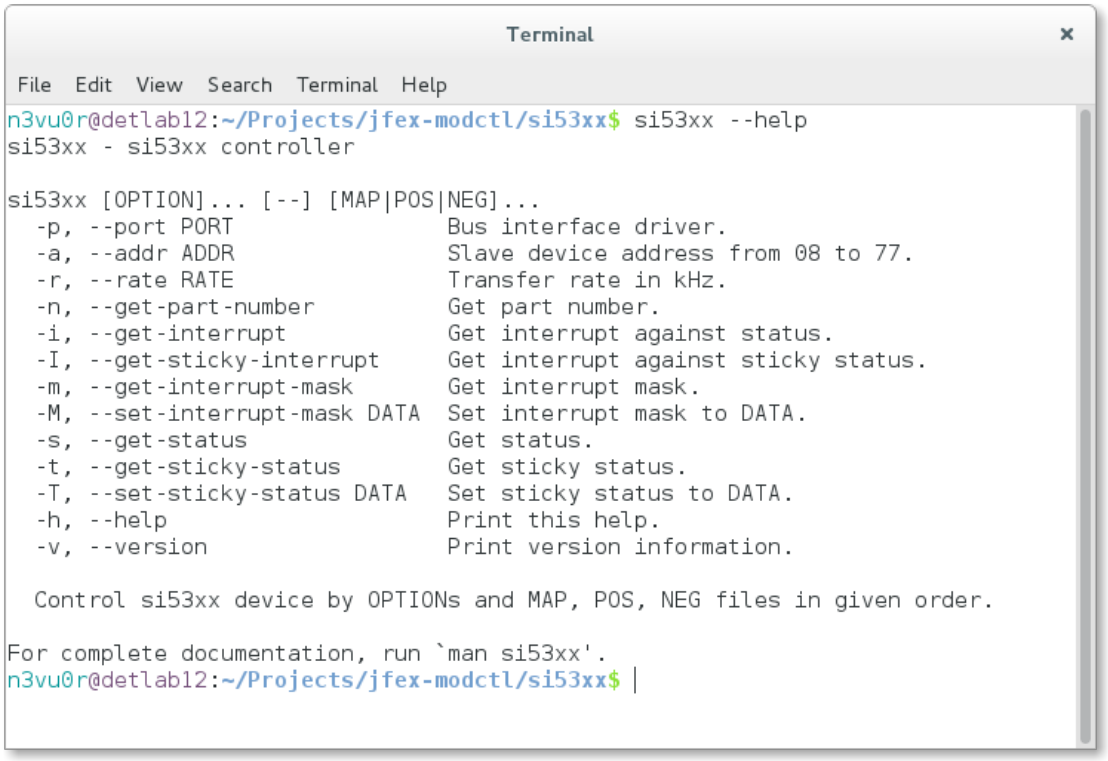


Figure 4.22: Usage of Clock Generator Controller

This application accepts multiple register map and transition map files as command line arguments labeled "[MAP|POS|NEG]..." and applies them one after another in given order. The square brackets indicate that specifying these files are optional since some basic built-in routines like reading the part number or status register of a device are implemented as well, specifiable by the "[OPTION]..." label.

The library abstracts away specifics of different I²C hardware interfaces by defining an intermediate but abstract C++ interface of two pure virtual functions[19] making different hardware interfaces interchangeable, as shown in line 44 and 45 of

[19] A pure virtual function requires a derived class to provide an implementation.

algorithm 4.14. In this way, the application can transparently be used with various internal or external I²C hardware by just implementing the abstract interface for each concrete hardware interface on demand. These two functions, `void get(byte addr, byte* data, byte size)` and `void set(byte addr, const byte* data, byte size)`, both accept a register address as first, an array of register values as second, and the number of these register values as third argument.

Algorithm 4.14 Hardware Abstraction

```
22 class si53xx {
23 public:
24     typedef unsigned char byte;
25     static const byte
26     PIN0 = 0x70,
27     PIN1 = PIN0 + 1;
28     static const byte
29     SYS_CAL = 1 << 0,
30     LOS_CLKIN = 1 << 2,
31     LOS_FDBK = 1 << 3,
32     PLL_LOL = 1 << 4;
33     bool set_register_map(std::string file);
34     std::string get_part_number();
35     bool get_interrupt();
36     bool get_sticky_interrupt();
37     byte get_interrupt_mask();
38     void set_interrupt_mask(byte data, byte mask = 0xFF);
39     byte get_status();
40     byte get_sticky_status();
41     void set_sticky_status(byte data, byte mask = 0xFF);
42     virtual ~si53xx() {};
43 protected:
44     virtual void get(byte addr, byte* data, byte size) = 0;
45     virtual void set(byte addr, const byte* data, byte size) = 0;
46 private:
47     byte rx(byte addr) { byte data; get(addr, &data, 1); return data; }
48     void tx(byte addr, byte data) { set(addr, &data, 1); }
49     void xx(byte addr, byte data, byte mask);
50     void pg(byte page) { tx(255, page & 0x01); }
51 };
```

For single byte mode, that is reading or writing only a single register, two helper functions, `byte rx(byte addr)` and `void tx(byte addr, byte data)`, are defined

in line 47 and 48 allowing a single register value to be returned and a single register value instead of an array of register values to be given as argument, respectively. A third helper function in line 50, `void pg(byte page)`, implements the already mentioned register page selection. These functions simplify code and increase its readability without sacrificing performance since they are not only declared but also defined in the C++ header file of the library allowing the compiler to optimize them away by substituting their definition in-line at their point of call [47]. In contrast, a fourth helper function, `void xx(byte addr, byte data, byte mask)`, is declared in line 49 of the C++ header file but defined in the C++ source file of the library due to its bigger code size, as shown in algorithm 4.15.

Algorithm 4.15 Read-Modify-Write Routine

```
23 void si53xx::xx(byte addr, byte data, byte mask) {
24     if (mask == 0xFF)
25         tx(addr, data);
26     else
27     if (mask != 0x00)
28         tx(addr, (rx(addr) & ~mask) | (data & mask));
29 }
```

This function makes use of the two single byte mode functions implementing the already motivated read-modify-write procedure by expecting an additional third argument for the register mask specifying the eight bits of an 1-byte sized register value intended to be written while ignoring the others. To ignore, actually means, to not change those bits. But since a whole byte must be written being the smallest transferable unit of the I²C bus, the old register value must be read firstly, then the bits intended to be written are modified, becoming the new register value finally being written to the device, hence read-modify-write. There are two special cases, a register mask of `0xFF` caught in line 24 and a register mask of `0x00` caught in line 27. Former means to write all eight bits, which line 25 reduces to a pure single write. Latter means to write no bits at all, which results in a skip of line 28 actually being the read-modify-write function. This line firstly reads the old register value `rx(addr)` and masks on the bits to be ignored with a binary “and” against the inverse register mask, `(rx(addr) & ~mask)`. Then it masks on the bits to be written of the new register value with a binary “and” against the proper register mask, `(data & mask)`. Now, a binary “or” safely merges the old bits to be ignored with the new bits to be written, `(rx(addr) & ~mask) | (data & mask)`. Finally, the modified register value is passed to the single write mode function.

Among single byte modes, burst modes can be performed as well by passing an array of multiple register values to the two hardware abstraction functions. Burst modes require a continuous register sequence, that is having incremental addresses. Burst writes additionally require register masks of `0xFF`, otherwise read-modify-writes are performed. For a continuous sequence it is not necessary to prepend the register address to each register value. Instead, only the address of the first register of the sequence is required since the device is capable of internally auto-incrementing addresses for I²C transactions of multiple register values. This drastically reduces the number of transactions, especially compared to single reads, being composed of two transactions in order to switch the I²C mode from write to read, former writing the register address and latter reading the register value, both beginning with the slave address of the device, as described in section 3.1.2. When this application applies register maps, it detects continuous `0xFF` masked register sequences and writes them in burst mode.

Chapter 5

Tests & Results

In this chapter the mezzanine card designed in chapter 3 is tested with the software packages developed in chapter 4. Which are the workflow kit Zed Tool (`z2l`), its helper tool `plug`, the CPU/FPGA file transfer application `fsio`, and the controller `si53xx` of the clock generator.

5.1 Booting the Operating System

With the help of the workflow kit of section §4.1, a Zed Tool project has been created to test if the mezzanine card is capable of booting into a terminal login prompt. This project was kept simple, having no user application software and an almost empty FPGA implementation, only connecting the eight switches of the mezzanine card to its eight LEDs. The boot process is monitored via an UART over USB terminal connection. The PicoZed exposes a signal labeled “DONE”, indicating the FPGA was successfully programmed. This signal is visualized by the mezzanine card with the left LED of the pair of blue lightning LEDs in figure 5.1.

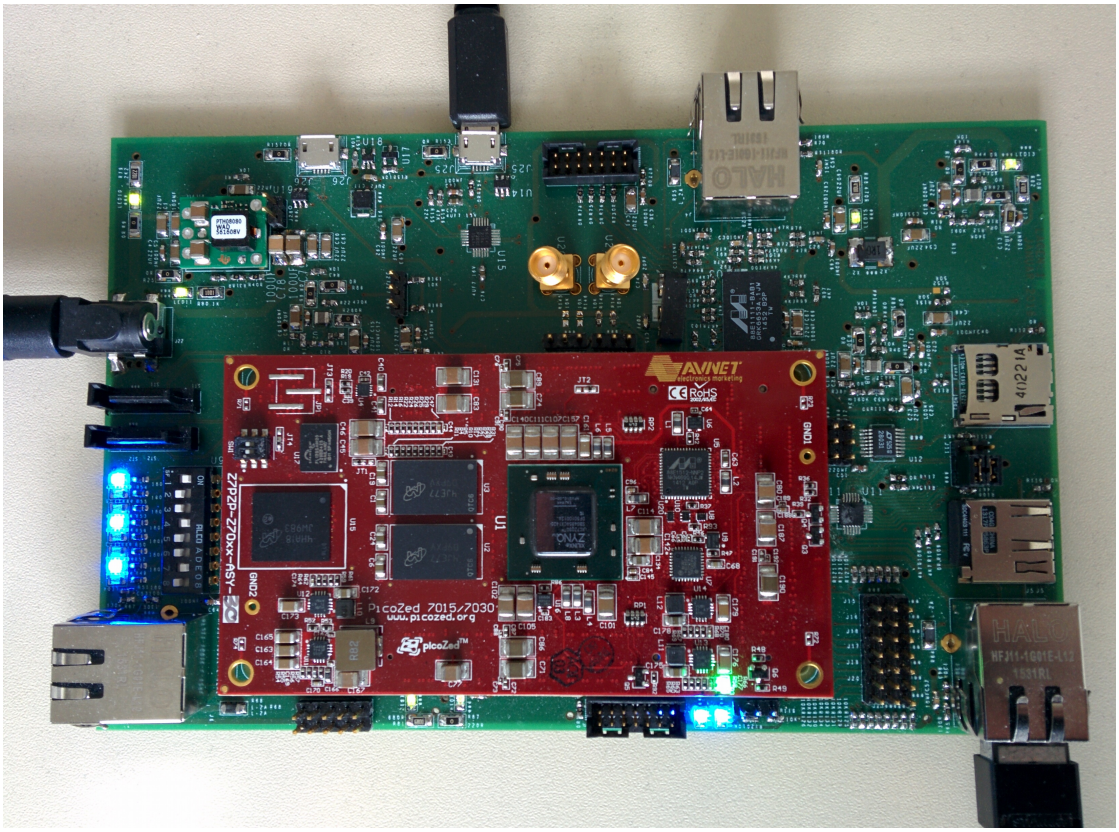


Figure 5.1: Booting the Mezzanine Card

The initial try to boot the mezzanine card successfully programmed the FPGA, the "DONE" LED was lightning and the array of eight LEDs correctly represented the eight switches on the left in figure 5.1. But no output of the boot process was printed on the terminal screen. Instead, the terminal connection has not been established by the USB-to-UART bridge of section 3.1.3 at all. Assuming the schematics of the mezzanine card to interface this chip are correct, bad solder connections might be the problem. Thus, this chip has manually been soldered again and the boot process was repeated. In fact, it solved the problem and the terminal connection was established. But there was still no output on the screen and surprisingly, the "DONE" LED was no more lightning. By default, the U-Boot allows the user to abort the boot process by pressing a key in the first four seconds. Now, having a connection established, it might be that a character is unintentionally sent to the terminal by a faulty connection. To confirm or disprove this, the UART module of the Zynq was disconnected from the USB-to-UART bridge by removing two $0\,\Omega$ resistor jumpers labeled "R149" and "R150" of its two wires, as shown in figure 5.2. Instead, it was connected to the RS-232 level translator by adding two $0\,\Omega$ resistor jumpers labeled "R114" and "R116".

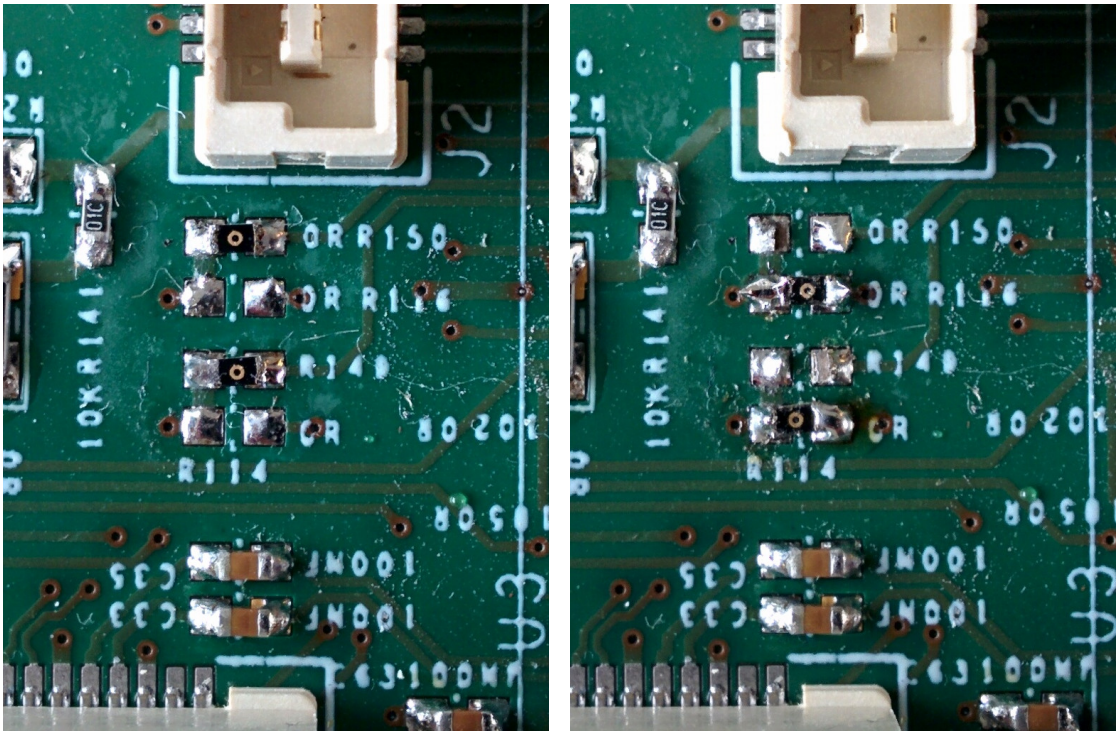


Figure 5.2: UART over USB – UART over RS-232

This did not affect the boot process, the FPGA was still no more programmed and no output on the UART over RS-232 terminal connection was seen. Another try was to completely disconnect the UART module of the Zynq by removing all jumpers, but again, it had no effect. Thus, it was not an unintentionally sent character aborting the boot process and these modifications have been undone to connect the USB-to-UART bridge again. The only modification done to the mezzanine card, after initially successfully programming the FPGA, was manually soldering the USB-to-UART bridge. Thus, it might be that this caused some mechanical stress to the mezzanine card possibly breaking some further badly soldered connections of other chips involved in the booting process like the SDIO port expander. Hence, another try has been made by power cycling the mezzanine card while pushing a finger onto the SDIO port expander chip to give some pressure on its pins. In fact, this solved the problem. The FPGA was successfully programmed again and output was finally printed on the screen. To allow the mezzanine card to boot without pushing a finger onto it, the SDIO port expander has been manually soldered again, as shown in figure 5.3. During cleaning the chip, its label was gone.

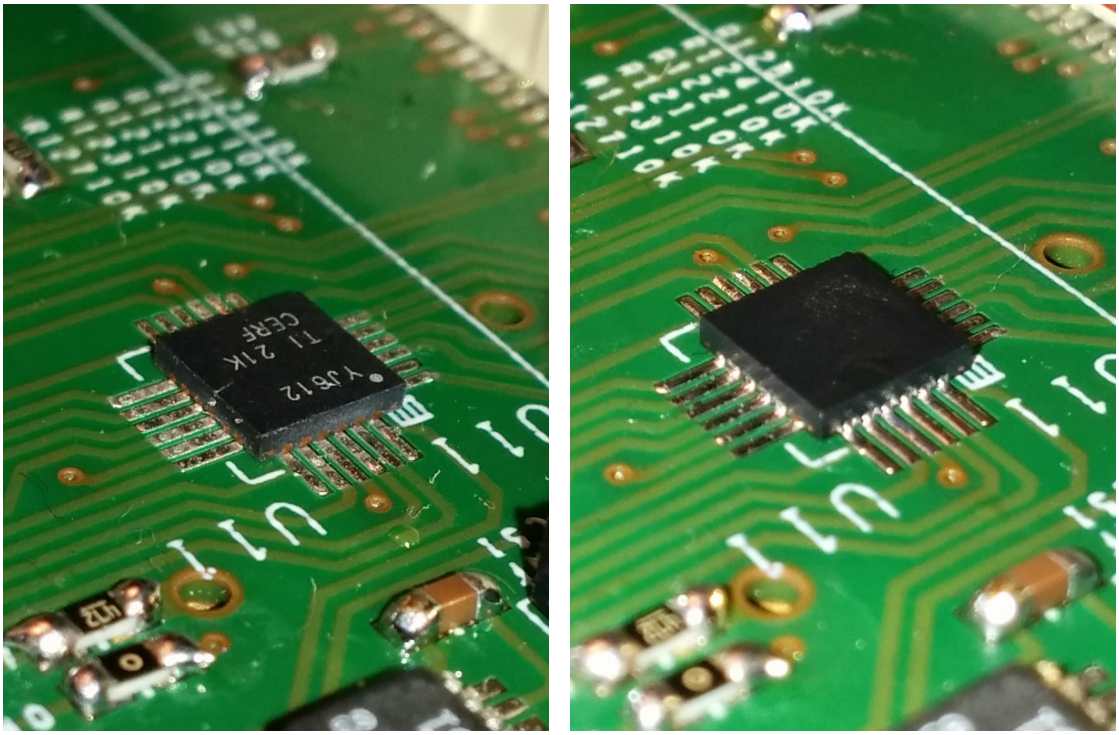


Figure 5.3: SDIO Port Expander [22]: Before & After

After having solved hardware related problems regarding the boot process, the output on the screen was inspected. The goal is to boot into a terminal login prompt after all the debugging information has been printed on the screen. But the boot process stopped at some point without an error message giving a hint about the problem. From the output it could be concluded that the FSBL was passed, the U-Boot loaded the kernel, and the kernel found and parsed the device tree while initializing one hardware module after the other until the boot process reproducibly stopped at the same point. The assumption was, that the device tree, being the last component, might cause the kernel to hang due to an invalid configuration of a hardware module. A device tree is typically described by a single device tree source file of file extension ".dts" and one or more device tree source include files of file extension ".dtsi", latter being included by former. These source files are compiled to a single device tree blob of file extension ".dtb", finally being parsed by the kernel. The device tree source include files are automatically generated by using the configurations of hardware modules contained within block design files of the Vivado Design Suite. Such a block design has been created and customized to describe the hardware modules used by the mezzanine card. Following customizations regarding the default configuration of the PicoZed have been done. The second SDIO module labeled "sdhci1" of the Zynq chip has been disabled since it is not used while the first one labeled "sd-

hci0" was already enabled and is connected to the SDIO port expander. The two I²C modules labeled "i2c0" and "i2c1" and the first UART module labeled "uart0" of the Zynq chip have been enabled. By default, its second UART module labeled "uart1" was already enabled and used for monitoring the boot process. Thus, it is expected to find corresponding entries of these modules in the device tree source include files. As shown in algorithm 5.1, these auto-generated files contain the required module descriptions. By default, the baud rate of the UART modules is 115.2 kbit/s and the frequency of the I²C modules is 400 kHz.

Algorithm 5.1 Hardware Module Descriptions

```
25 &i2c0 {
26     clock-frequency = <400000>;
27     status = "okay";
28 };
29 &i2c1 {
30     clock-frequency = <400000>;
31     status = "okay";
32 };
```

```
42 &sdhci0 {
43     status = "okay";
44     xlnx,has-cd = <0x1>;
45     xlnx,has-power = <0x0>;
46     xlnx,has-wp = <0x0>;
47 };
48 &uart0 {
49     current-speed = <115200>;
50     device_type = "serial";
51     port-number = <0>;
52     status = "okay";
53 };
54 &uart1 {
55     current-speed = <115200>;
56     device_type = "serial";
57     port-number = <1>;
58     status = "okay";
59 };
```

Additionally, the kernel must know to which UART module it should print the boot messages. As shown in algorithm 5.2, this is defined by creating an alias “serial0” and point it to the appropriate UART module, in this case “uart1”.

Algorithm 5.2 Hardware Module Assignments

```
13      aliases {
14          serial0 = &uart1;
15          ethernet0 = &gem0;
16          spi0 = &qspi;
17      };
```

But an alias entry for the additionally enabled UART module labeled “uart0” was missing. Thus, it has been manually added to the single device tree source file of file extension “.dts”, not to be confused with the “.dtsi” files which will be overwritten the next time their auto-generation is triggered. This “.dts” file is shown in algorithm 5.3 and can be edited by invoking the Zed Tool target `fw.edit.dts`. The default I²C frequencies of $400\,\mathrm{kHz}$ have been changed to $100\,\mathrm{kHz}$ as well, lowering the capacitive requirements of the I²C chains.

Algorithm 5.3 Hardware Module Modifications

```
1   /dts-v1/;
2   /include/ "system-conf.dtsi"
3   / {
4       aliases {
5           serial1 = &uart0;
6       };
7   };
8   &i2c0 {
9       clock-frequency = <100000>;
10  };
11  &i2c1 {
12      clock-frequency = <100000>;
13  };
```

Having added this missing alias for the additionally enabled UART module solved the problem of a stopping boot process, the mezzanine card was successfully booting into a terminal login prompt. Thus, this modification has been saved to the PetaLinux source folder managed by Zed Tool by invoking its target `fw.save`.

To speed up the boot process, the previously mentioned delay of four seconds can be disabled by editing a C code header file of the U-Boot with the help of Zed Tool by invoking its target `fw.edit.ubl` and redefining the C Preprocessor (CPP) constant `CONFIG_BOOTDELAY` from value `4` to `-2`, interpreted as completely disabling the possibility of aborting the boot process. In analogy to the device tree sources, this header file is not auto-generated and thus will not be overwritten, but it includes other auto-generated header files providing hardware information to the U-Boot. Another optimization has been made to not only save time but to also prevent potential IP address[1] conflicts in case the final MAC address[2] of the mezzanine card will be determined at a later stage. By default, the U-Boot initializes the Ethernet module with a MAC address fixed at compile time. This initialization triggers an Ethernet procedure called auto-negotiation by which common transmission parameters like the speed mode are determined. This takes time and is of no use, since the kernel is supposed to manage Ethernet connectivity. The U-Boot does not provide a single CPP constant to disable this feature. Instead, a workaround is to tell the U-Boot that this Ethernet module does not exist by revoking all CPP constants related to this module, again with the help of the Zed Tool target `fw.edit.ubl` followed by `fw.save` to save these modifications.

5.2 Testing the CPU/FPGA Communication

The feedback synchronized CPU/FPGA communication of section §4.2 is tested with the mezzanine card of section §3.2. The Zynq chip of the PicoZed is of speed grade `-1`. For this speed grade the product guide [43] of the AXI GPIO IP Core states a maximum clock frequency of $180\,\mathrm{MHz}$ for the AXI GPIO core which is used in this test. The embedded operating system PetaLinux 2015.4 was targeted by the cross compiler `arm-linux-gnueabi-gcc` [59] of version 4.9.2 with optimization switch `-O3`. Firstly, the data integrity of the proposed protocol has been tested. Then, transfer rate measurements were performed. Besides a connected power supply, an Ethernet connection to the CPU of the mezzanine card was used to control the software via multiple remote terminals using the tool `ssh` [39]. The host name of the mezzanine card is “jfex”, displayed on each terminal prompt. Additionally, an UART

[1] An Internet Protocol (IP) address can be assigned in dependence of a MAC address.
[2] A media access control (MAC) address is a unique identifier of a network interface.

over USB connection was used to monitor the reboot process in case the FPGA must be reprogrammed due to modifications to the slave implementation of the protocol. All this was done with the help of the workflow kit Zed Tool described in section §4.1.

5.2.1 Data Integrity Verification

This test uses two remote terminals. One for writing a test vector file to the FPGA and another for reading it back. Afterwards, their checksums are compared. The FPGA instantiates two channels of equal width, one for reading data and the other for writing data back, as shown in algorithm 5.4. The signal names ending in "0" are the ones of the writing channel. The read data is locally stored in the signal vector "data".

Algorithm 5.4 Two Channels as Echo Check

```
191     ctl: process(clk180)
192     begin
193         if rising_edge(clk180) then
194             if req0 then
195                 if full and not ack0 then
196                     dat0 <= data;
197                     full <= '0';
198                     ack0 <= '1';
199                 end if;
200             else
201                 ack0 <= '0';
202             end if;
203             ack1 <= '0';
204             if req1 then
205                 if not full then
206                     data <= dat1;
207                     full <= '1';
208                     ack1 <= '1';
209                 end if;
210             end if;
211         end if;
212     end process ctl;
```

As shown in figure 5.4, a test vector of $20000000 \cdot 1 \cdot 4\,\mathrm{B} = 80\,\mathrm{MB}$ was generated. This is roughly the maximum file size of a bit stream file for a processor FPGA used by the jFEX. The channel description file is dumped on the screen. It consists of one

handshake and four data maps of 4 B each. Finally, the file transfer application `fsio` has been executed. It established the previously dumped communication channel and requested the FPGA to write data. The expected file size to be read, that is the size of the test vector, is given as command line argument option `-e`. The test vector was read back to file "get.tv" while transfer statistics were stored to file "get.sv".

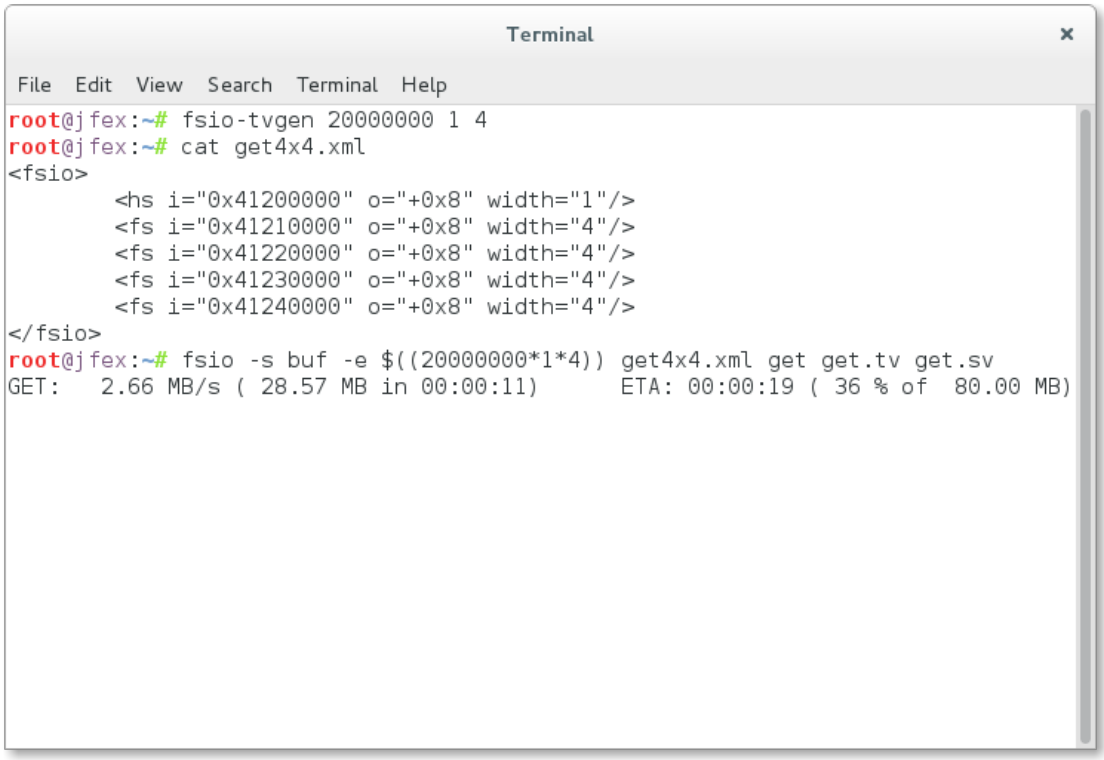


Figure 5.4: Reading Back the Test Vector

The file transfer started as soon as the other terminal has executed the `fsio` application for writing the generated test vector "20000000x1x4.tv", as shown in figure 5.5. For writing the file to the FPGA, a separate channel has been established. Its description file is dumped on the screen. It is of equal width and uses different mapping addresses. After roughly 31 s the transfer has finished with an averaged transfer rate of roughly 2.6 MB/s. For more precise values, the statistic vectors "get.sv" and "put.sv" are meant to be inspected. The final command "sha1sum" calculated the checksums of the written file "20000000x1x4.tv" and the read file "get.tv". Both files had identical checksums indicating a successful file transfer of no data corruption.

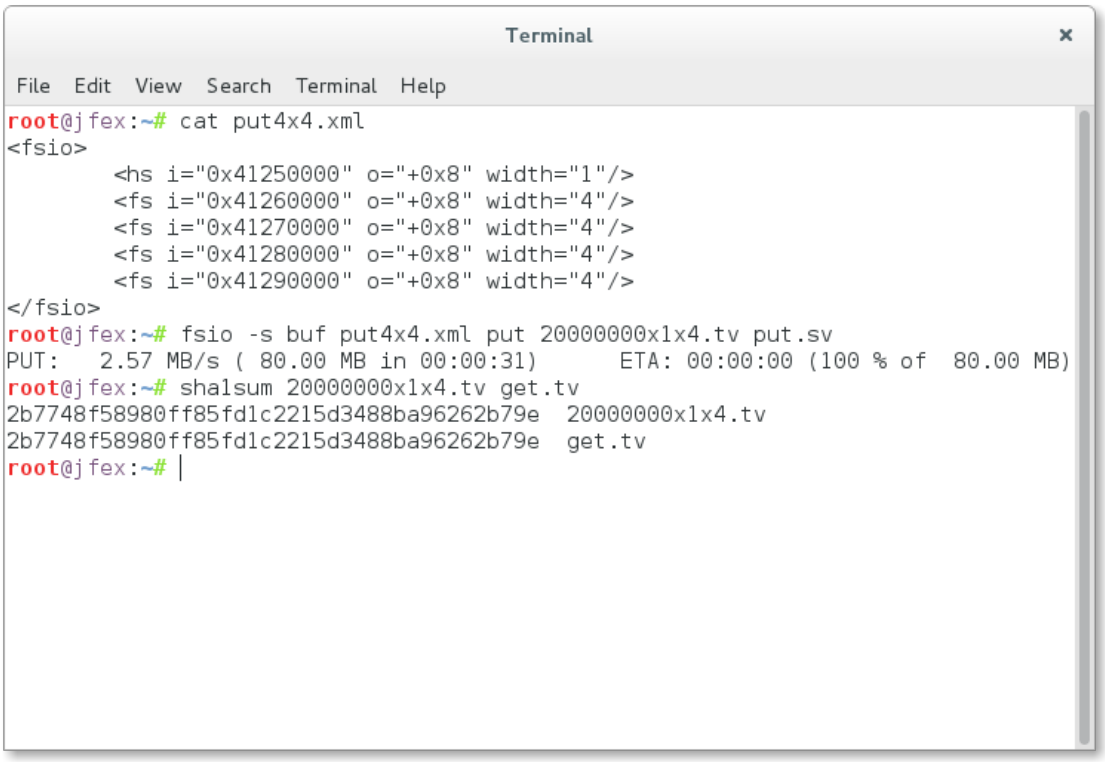


Figure 5.5: Writing the Test Vector

After the first successful test, it has been automated by a script. This script repeatedly executed file transfers and compared written with read back files. It only stops when the checksums mismatch. After roughly two days and $460.8\,\mathrm{GB}$ transferred in total, this script has been aborted manually. No single data corruption was observed.

5.2.2 Write Rate Measuring

Previous tests already gave a hint of several $\mathrm{MB/s}$ for the magnitude of transfer rates. But the CPU was busy with transferring data in both directions instead of measuring only one transfer at a time. Now, the assumption of section 4.2.2 that more data maps per handshake map might increase the throughput rate until saturation was precisely measured by making use of the statistic vectors the `fsio` application is capable to generate. The first ten rows of such a statistic vector are dumped in figure 5.6. Each row contains the transfer rate in $\mathrm{B/s}$ averaged over the time span the application buffer is being filled. When full, it is emptied and the averaged transfer rate is written to the next row. The default buffer size was used, that is 16384 times a channel width. For more statistics, a bigger test vector of $4 \cdot 80\,\mathrm{MB} = 320\,\mathrm{MB}$

was generated. That is roughly the sum of bit stream sizes for all four processor FPGAs. Since the operating system mounts its root file system in RAM, it is ensured that a possible slower read rate of the SD card cannot distort the measurement. For the same reason transfer statistics were only displayed at the end of the file transfer by the option `-s` to not waste CPU cycles.

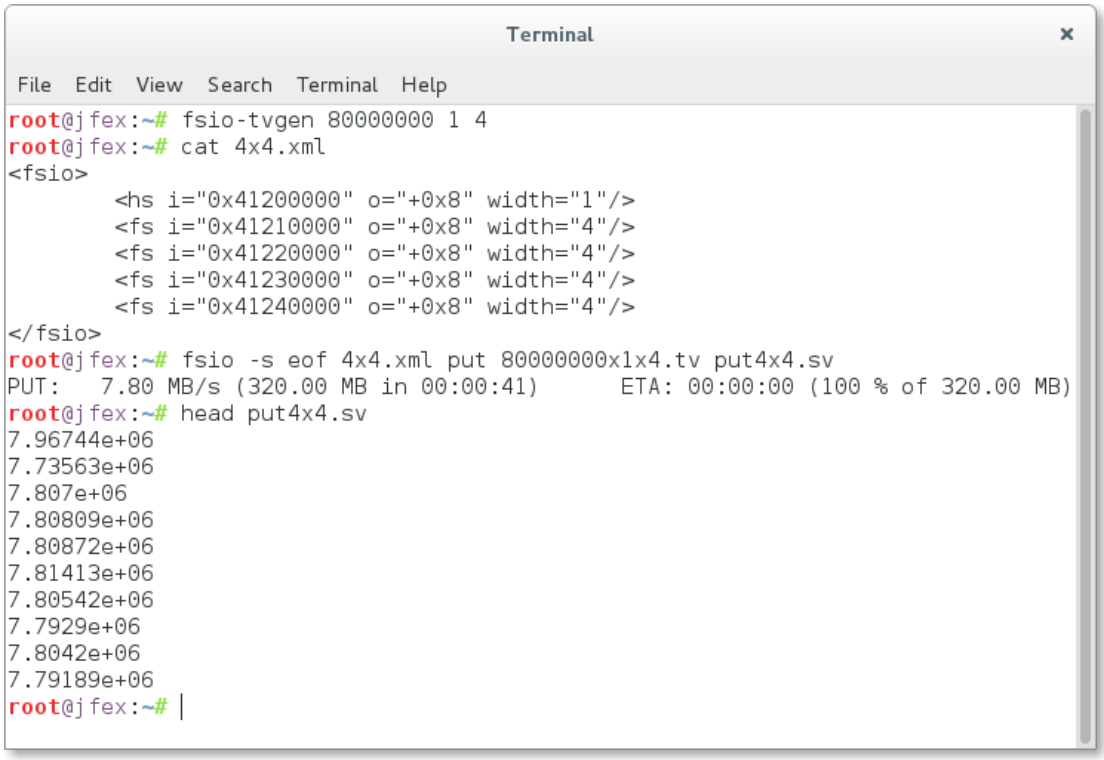


Figure 5.6: Write Rate Measuring

To test various channel widths, 32 different channel description files from width of $1 \cdot 4\,\mathrm{B} = 4\,\mathrm{B}$ to $32 \cdot 4\,\mathrm{B} = 128$ have been created with the largest one of 32 data maps. The test vector of $320\,\mathrm{MB}$ was written to each of these differently sized channels. The data read by the FPGA was discarded. The mean values of the transfer rates of the statistic vectors along with their standard deviations as uncertainties have been calculated and are listed in table 5.1.

Maps	Channel Width (B)	Rate (MB/s)	Rate Uncertainty (MB/s)
1	4	4.540	0.006
2	8	6.288	0.016
3	12	7.231	0.008
4	16	7.798	0.015
5	20	8.095	0.038
6	24	8.452	0.009
7	28	8.672	0.010
8	32	8.827	0.010
9	36	8.986	0.016
10	40	9.095	0.013
11	44	9.135	0.012
12	48	9.203	0.015
13	52	9.314	0.013
14	56	9.386	0.013
15	60	9.452	0.013
16	64	9.472	0.014
17	68	9.461	0.015
18	72	9.480	0.015
19	76	9.555	0.018
20	80	9.594	0.019
21	84	9.630	0.017
22	88	9.652	0.017
23	92	9.679	0.017
24	96	9.687	0.018
25	100	9.717	0.018
26	104	9.733	0.018
27	108	9.737	0.019
28	112	9.734	0.019
29	116	9.735	0.019
30	120	9.734	0.020
31	124	9.736	0.020
32	128	9.733	0.021

Table 5.1: Write Rate vs. Channel Width

From a map count of 1 to 8 the write rate is almost doubled with a factor of

$$\frac{(8.827 \pm 0.010)\,\mathrm{MB/s}}{(4.540 \pm 0.006)\,\mathrm{MB/s}} = 1.944 \pm 0.004$$

but from a map count of 8 to 32 the increase goes into saturation with a factor of

$$\frac{(9.733 \pm 0.021)\,\mathrm{MB/s}}{(8.827 \pm 0.010)\,\mathrm{MB/s}} = 1.103 \pm 0.003$$

as visualized in figure 5.7 showing the the transfer rate in $\mathrm{MB/s}$ against the channel width in B confirming the assumption that more data maps per handshake map might increase the throughput rate until saturation.

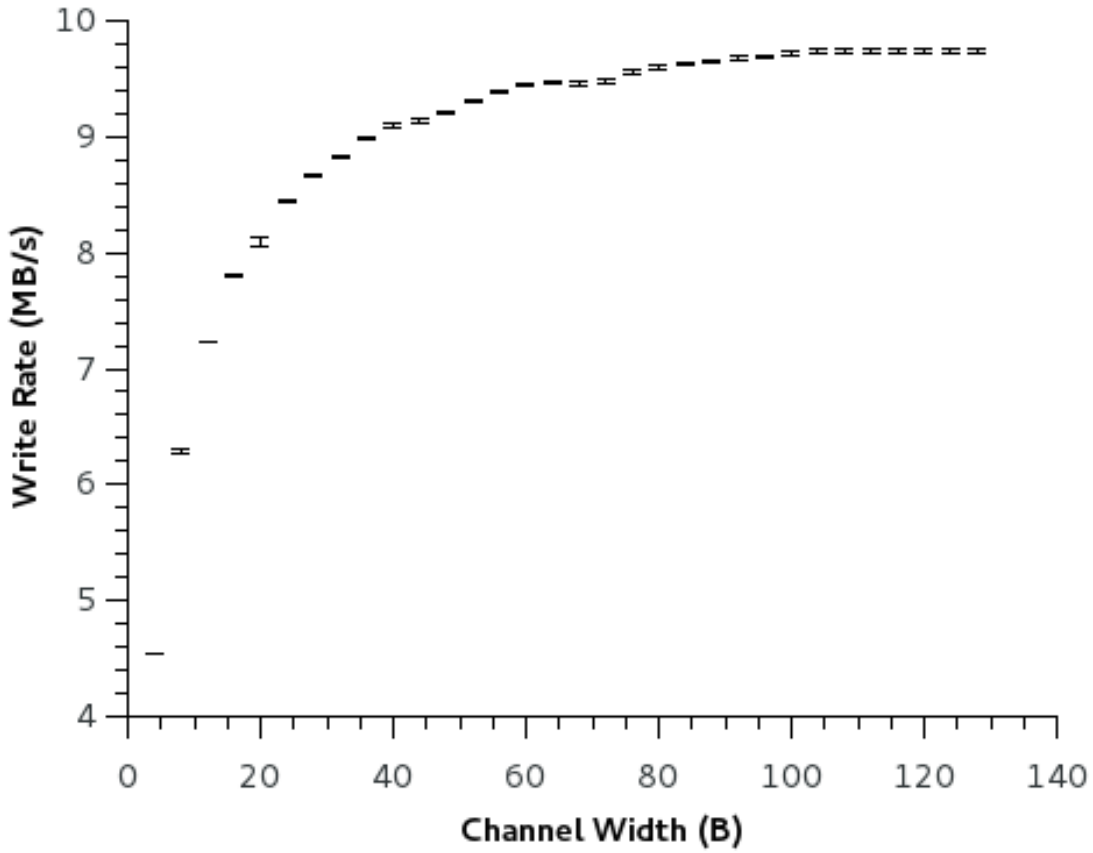


Figure 5.7: Write Rate vs. Channel Width

The maximum factor of increase relative to 1 data map of

$$\frac{(9.737 \pm 0.019)\,\mathrm{MB/s}}{(4.540 \pm 0.006)\,\mathrm{MB/s}} = 2.145 \pm 0.006$$

is observed with 27 data maps.

These results can be considered a success regarding the rough requirement of programming the processor FPGAs in a reasonable amount of time. When instantiating

a channel of 4 or 8 data maps, all four bit streams can be transferred in less than a minute, $(41.04 \pm 0.08)\,\mathrm{s}$ or $(36.25 \pm 0.05)\,\mathrm{s}$, respectively. The write rate will not be slowed down when choosing an appropriate SD card since the SDIO port expander and the Zynq support both bus speeds [22, 60], Normal Speed of up to $12.5\,\mathrm{MB/s}$ and High Speed of up to $25\,\mathrm{MB/s}$ [61], exceeding the maximum write rate from CPU to FPGA of $(9.737 \pm 0.019)\,\mathrm{MB/s}$ with 27 data maps. The FPGA of the Zynq will forward one bit stream after the other to the corresponding processor FPGA of the jFEX over one of its seven so-called configuration interfaces in order to program it [10]. One interface is called Slave SelectMAP and can operate at a clock rate of up to $125\,\mathrm{MHz}$ [62], resulting in a transfer rate of up to $125\,\mathrm{MB/s}$ when choosing a bus width of $8\,\mathrm{bits}$. Since both the transfer rate of reading from an appropriate SD card and the transfer rate of writing to the processor FPGAs exceed the maximum write rate from CPU to FPGA, the amount of time required to program the processor FPGAs depends on the number of data maps only.

5.2.3 Read Rate Measuring

In contrast to the write rate measuring of section 5.2.2, it is not enough to have 32 different channel description files used with actual only one channel of the maximum width of 32 data maps, since when reading from the FPGA, it would always compare the fed back data of all maps but the number of maps being feed back by the CPU varies due to different description files. Instead of regenerating the bit stream for the FPGA for each channel width, an additional channel of one data map of size $1 \cdot 4\,\mathrm{B} = 32\,\mathrm{bit}$ was instantiated. Each of its 32 signals is used to dynamically enable comparison for each of the 32 maps at run time. Thus, before a read measuring started, the channel width has been configured appropriately by writing a $4\,\mathrm{B}$ sized binary file named “width” appended by the number of maps, as shown in figure 5.8.

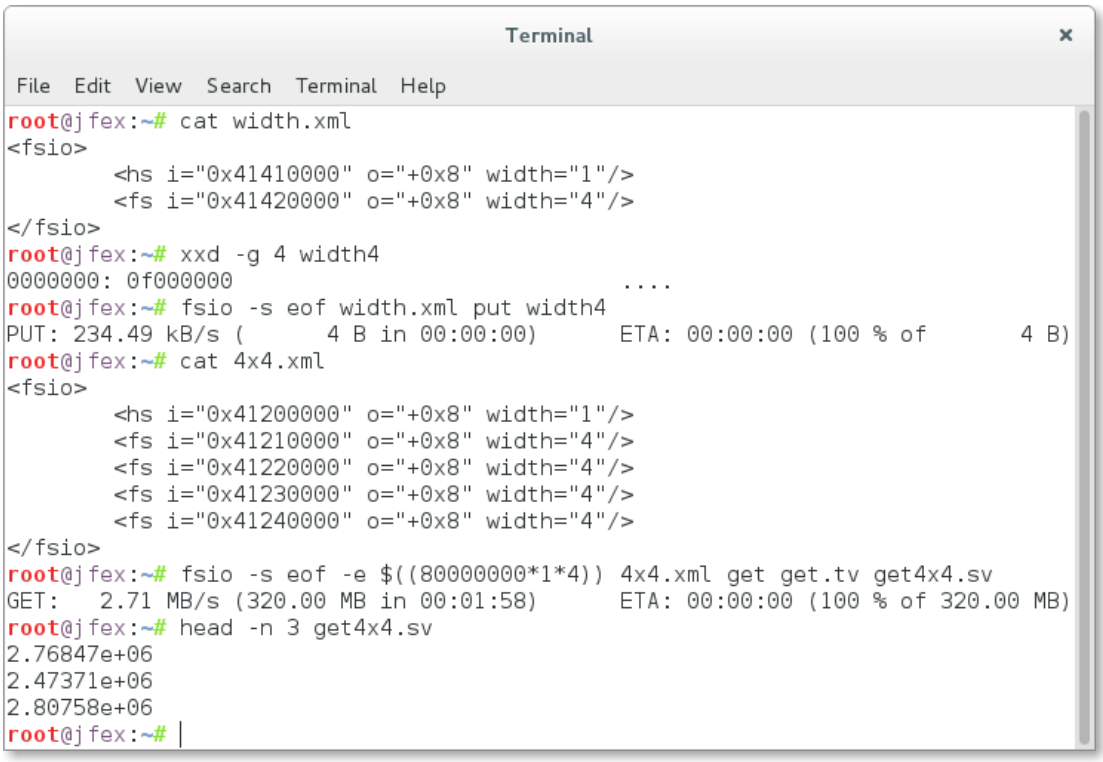


Figure 5.8: Read Rate Measuring

A test vector of 320 MB was generated by the FPGA on the fly. It was read from each of these differently sized channels and written to the file system mounted in RAM. The mean values of the transfer rates of the statistic vectors along with their standard deviations as uncertainties have been calculated and are listed in table 5.2.

Maps	Channel Width (B)	Rate (MB/s)	Rate Uncertainty (MB/s)
1	4	0.855	0.057
2	8	1.567	0.091
3	12	2.184	0.114
4	16	2.722	0.123
5	20	3.191	0.131
6	24	3.605	0.130
7	28	3.979	0.126
8	32	4.337	0.118
9	36	4.611	0.110
10	40	4.894	0.102
11	44	5.052	0.086
12	48	5.319	0.077
13	52	5.507	0.063
14	56	5.676	0.058
15	60	5.862	0.042
16	64	6.049	0.030
17	68	6.167	0.020
18	72	6.276	0.009
19	76	6.405	0.008
20	80	6.534	0.015
21	84	6.626	0.022
22	88	6.738	0.026
23	92	6.848	0.030
24	96	6.958	0.031
25	100	7.017	0.031
26	104	7.092	0.032
27	108	7.195	0.030
28	112	7.266	0.027
29	116	7.335	0.027
30	120	7.385	0.025
31	124	7.438	0.025
32	128	7.545	0.021

Table 5.2: Read Rate vs. Channel Width

From a map count of 1 to 12 the read rate is roughly sextupled with a factor of

$$\frac{(5.319 \pm 0.077)\,\mathrm{MB/s}}{(0.855 \pm 0.057)\,\mathrm{MB/s}} = 6.221 \pm 0.425$$

but from a map count of 12 to 32 the increase goes into saturation with a factor of

$$\frac{(7.545 \pm 0.021)\,\mathrm{MB/s}}{(5.319 \pm 0.077)\,\mathrm{MB/s}} = 1.418 \pm 0.021$$

as visualized in figure 5.9 showing the the transfer rate in $\mathrm{MB/s}$ against the channel width in B confirming the assumption that more data maps per handshake map might increase the throughput rate until saturation.

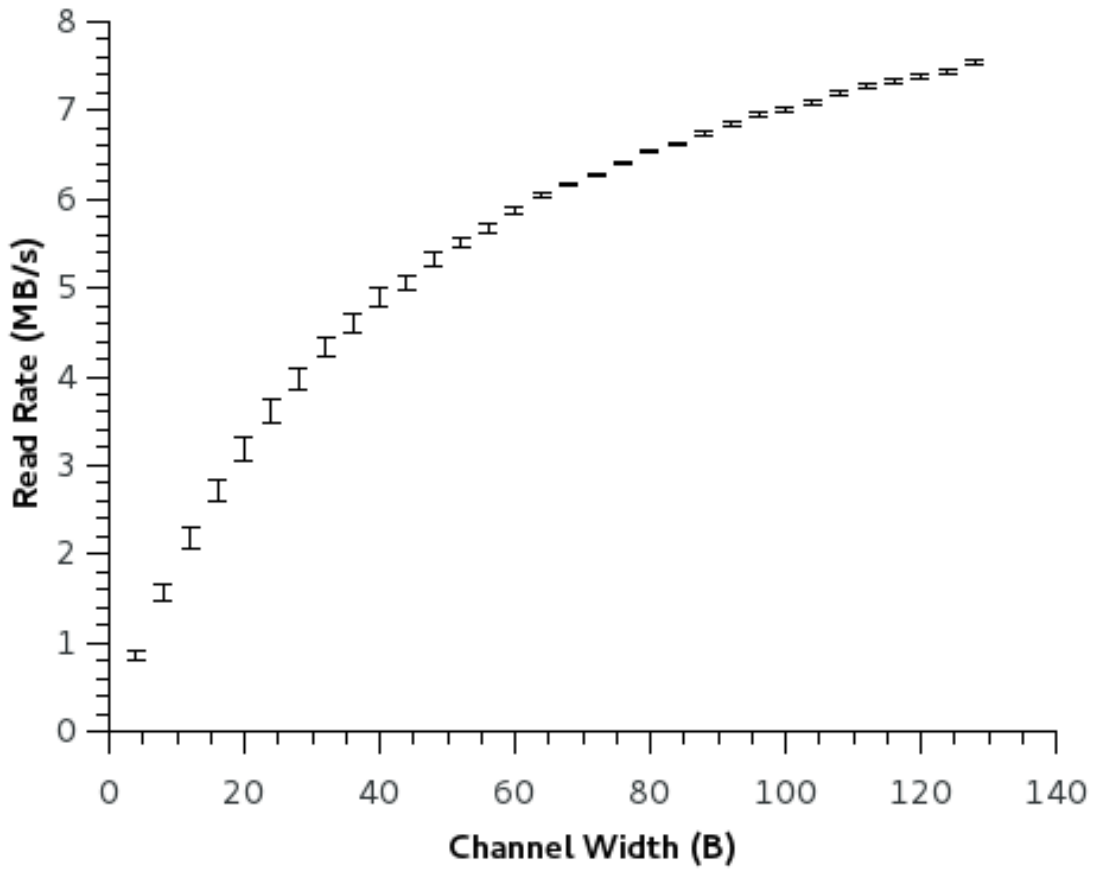


Figure 5.9: Read Rate vs. Channel Width

The maximum factor of increase relative to 1 data map of

$$\frac{(7.545 \pm 0.021)\,\mathrm{MB/s}}{(0.855 \pm 0.057)\,\mathrm{MB/s}} = 8.825 \pm 0.589$$

is observed with 32 data maps.

Though this maximum factor of increase compared with the one of the write rate of 2.145 ± 0.006 is by factor 4.116 ± 0.275 higher, the read rates are overall lower then the

write rates. This might be due to the fact that the CPU feeds data back to the FPGA alternately with the handshake polling instead of continuously like the FPGA does, resulting in a higher delay per handshake reducing the effective throughput rate.

5.3 Testing the I²C Communication

While the mezzanine card of section §3.2 was being produced, I²C communication as described in section 3.1.2 was firstly tested with different hardware. This test confirms functionality of an I²C master module by connecting it to an I²C slave device of testing purpose only. In this way, a possible erroneous I²C transaction cannot damage a slave device like the clock generator of section §4.3 by accidentally writing faulty data to wrong registers. As I²C master module, an USB to serial/parallel break-out module called UM232H-B [63] from Future Technology Devices International Ltd. (FTDI) was chosen. Besides other communication standards it supports I²C in master mode. The functionality of this mode is implemented by a library called `libmpsse` [64]. It is based on a general-purpose library for various FTDI chips called `libftdi` [65] which interfaces the device via USB with the help of a common library called `libusb` [66]. Due to its several debugging facilities, a microcontroller development board called RN-Control [67] was chosen as I²C slave device. It is populated with an ATmega32 [68] chip, an AVR microcontroller from Atmel. This chip features a hardware I²C slave interface referred to as Two Wire Interface (TWI) by its data sheet. This family of microcontrollers is programmed via USB by so-called in-system programmers (ISP). That means, it is not required to unplug it from the development board in order to program it. The programmer to be used is called USBasp [69]. The whole setup is shown in figure 5.10 with the programmer on the top left, the I²C master module on the bottom left, and the development board on the right. The two I²C wires, the data line SDA in yellow and the clock line SCL in blue, are connected along with the ground wire GND in green from the master module to the slave device. The master module provides separate pins for data input and data output, “D2” and “D1”, respectively. Since the data line SDA is input and output as well, both pins must be connected to it. The clock signal is generated on pin “D0” and ground is labeled “GND”. The development board exposes SDA on pin 2, SCL on pin 1, and GND on pin 9 of its port labeled “C”. Switch 1 and 2 are off to disconnected SDA and SCL from LED 2 and LED 1, respectively. Though, the multi-purpose FTDI chip of the master module can operate the pins “D1” and “D0” as open drains, the library `libmpsse` drives them actively high to $3.3\,\mathrm{V}$. But the I²C slave module of the microcontroller operates at $5\,\mathrm{V}$ while the inputs of the master

module are $5\,\mathrm{V}$ tolerant. Thus, the internal $10\,\mathrm{k\Omega}$ pull-up resisters of the development board are bridged with external ones of $1\,\mathrm{k\Omega}$ against the USB supply voltage of $5\,\mathrm{V}$ to pull up the $3.3\,\mathrm{V}$ more aggressively.

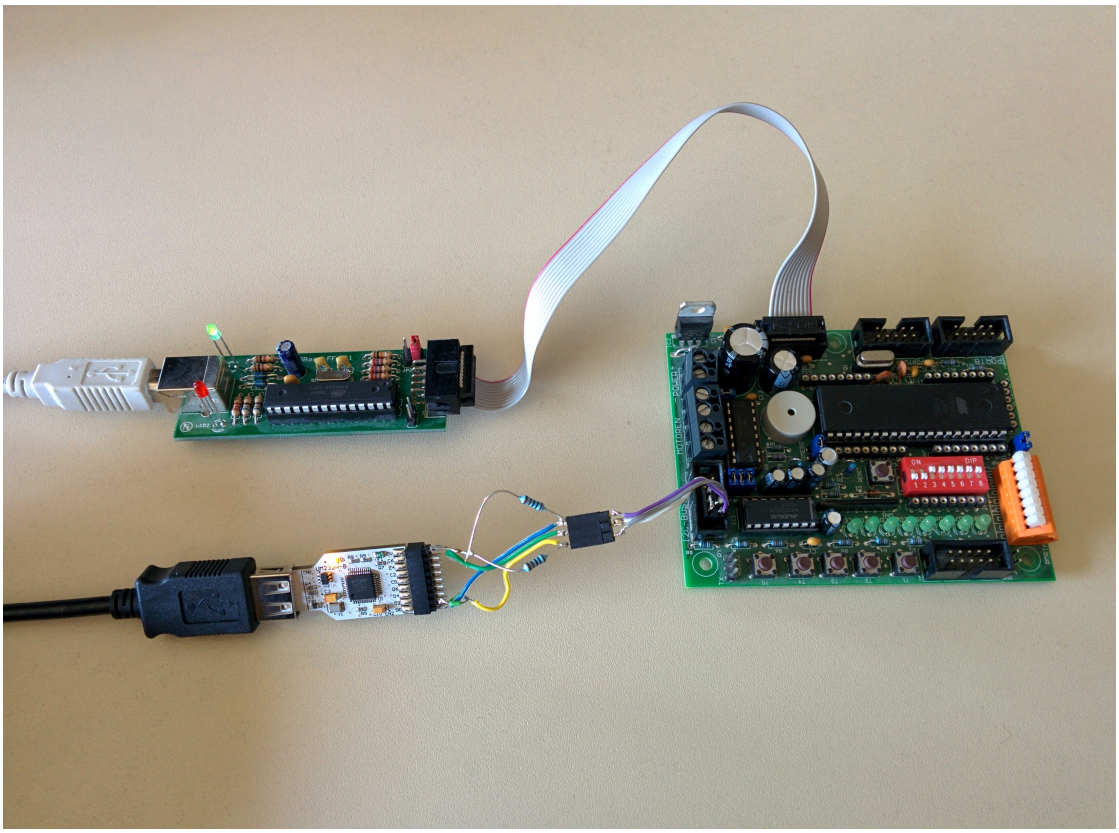


Figure 5.10: I²C Communication Test [63, 67, 69]

For testing purpose only, a C++ application was written interfacing the I²C master module by making use of the library `libmpsse`. It accepts a variable number of hexadecimal pairs as command line arguments each representing a byte. The first byte is the address of the slave device the following data bytes are supposed to be written to. It expects the slave device to write back the same number of data bytes and dumps them on the screen. The implementation of the slave device is written in C++ as well, being totally event-driven regarding master write (MW) and master read (MR) requests. When the master requests the slave to read its data, the slave stores it in a fixed sized ring buffer and acknowledges each byte until its buffer is full signifying the master that it cannot store any more data bytes. When the master requests the slave to write data back, it starts reading its ring buffer from the position of the firstly stored data byte of the last write transaction. The master has to signify the slave to stop writing back by not acknowledging the last data byte.

This is a so-called echo check, what is written is expected to be read back and echoed on the screen for confirmation.

When executing the test application with the slave address followed by one data byte as command line arguments, it successfully dumped the looped back data byte on the screen. That means, interfacing and initializing the master module is functioning. But subsequent invocations of the application resulted in a data byte of value zero. This was reproducible after the slave device has been reset. Again, only the first invocation returned the correct data byte while subsequent ones were of value zero. Since the first byte has always been transferred correctly, it was not a problem of the I²C hardware or its wiring. Thus, it had to be a faulty implementation. For further debugging information, the LED array of the development board is well suited to represent small integer numbers with little programming effort. To monitor the slave device regarding a write request of the master, the calls of the slave write routine were counted and visualized with help of the LEDs by switching on one after the other each call. For normal operation, each invocation of the application with one data byte as argument would switch on one additional LED. Surprisingly, after the first invocation, two LEDs were lightning. Thus, the zero valued bytes have been caused by reading them from the wrong position of a zero value initialized ring buffer since a call of the slave write routine increments a position counter running away from its supposed position due to the additional unintended call of the slave write routine. It turned out that the master read routine of the library `libmpsse` acknowledges all requested bytes including the last one. Acknowledging a byte in master read mode is interpreted as a request to write another byte in slave write mode. This caused the additional routine call. Instead of having an additional argument defined for the master read routine to prevent acknowledging the last byte, the library `libmpsse` defines two additional functions, one for disabling and one for enabling acknowledgement of all subsequent bytes. In order to read a variable number of data bytes, two calls are necessary. The first call reads all except the last byte, then acknowledgment is disabled and the last byte is read by the second call. After having this corrected, all was functioning as expected. Also multiple data bytes as command line arguments were successfully transferred. The I²C clock speed was being increased up to roughly $200\,\mathrm{kbit/s}$ until transactions began to fail, giving a safety margin for this setup of roughly factor 2 regarding the clock speed for the standard mode of $100\,\mathrm{kbit/s}$.

After having confirmed the functionality of the I²C master module, this application was used as template to implement the abstract hardware interface of the library `libsi53xx` of section §4.3. By choosing this implementation with a command line

argument of the application `si53xx`, it can control the clock generator via this I²C master module.

5.4 Controlling the Clock Generator

Controlling the clock generator was tested with the help of the Si5338 evaluation board from Silicon Labs, shown in figure 5.11. It exposes its input and output clocks via SMA connectors. The ClockBuilder Desktop Software from Silicon Labs can connect to the clock generator populated in the center of this board via USB, serving also as power supply. A microcontroller unit (MCU) of the board does the USB to I²C translation. The I²C master of the MCU can be disconnected from the clock generator by removing the two jumpers, one for the I²C data and the other for the I²C clock, of the header labeled “J19”. In this way, an external I²C master can be injected by plugging the yellow and blue colored wires of the connector on the left of figure 5.11 to this header while its green colored ground is wired to the connector labeled “GND”. Four status LEDs are present on the top left labeled “RDY”, “I2C”, “USB”, and “INTR” from top to bottom, the first three light green and the last one red. “RDY” indicates the board is operating normally. “I2C” and “USB” each visualize activity of their bus. “INTR” lights if the clock generator triggered its interrupt pin customizable by an interrupt mask as described in section §4.3.

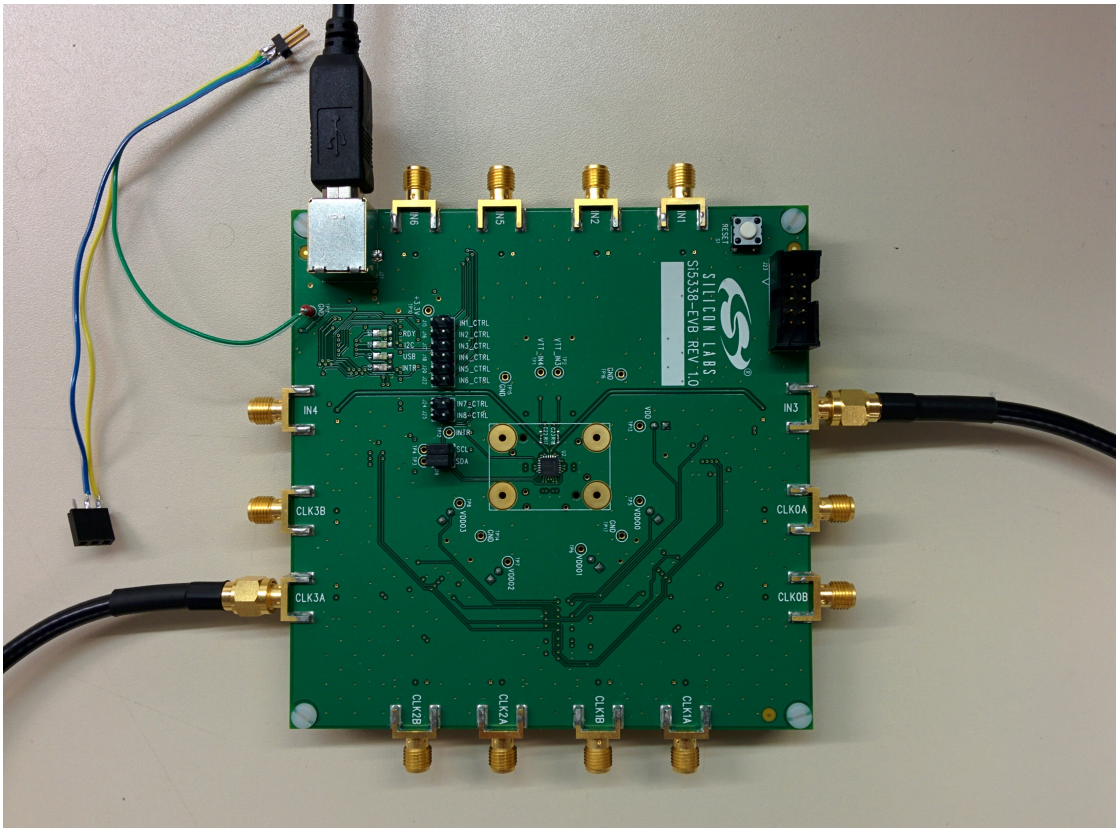


Figure 5.11: Si5338 Evaluation Board [54]

5.4.1 Built-in Routines

Firstly, the built-in routines of the application `si53xx` of section §4.3 were tested with an external I²C module, which itself has previously been tested in section §5.3. As shown in figure 5.12, no clocks are connected for this setup. Since the default configuration of the clock generator expects a differential input clock on the connectors “IN1” and “IN2”, it has thrown the error flag “LOS_CLKIN”, loss of clock input, which caused the red interrupt LED to light.

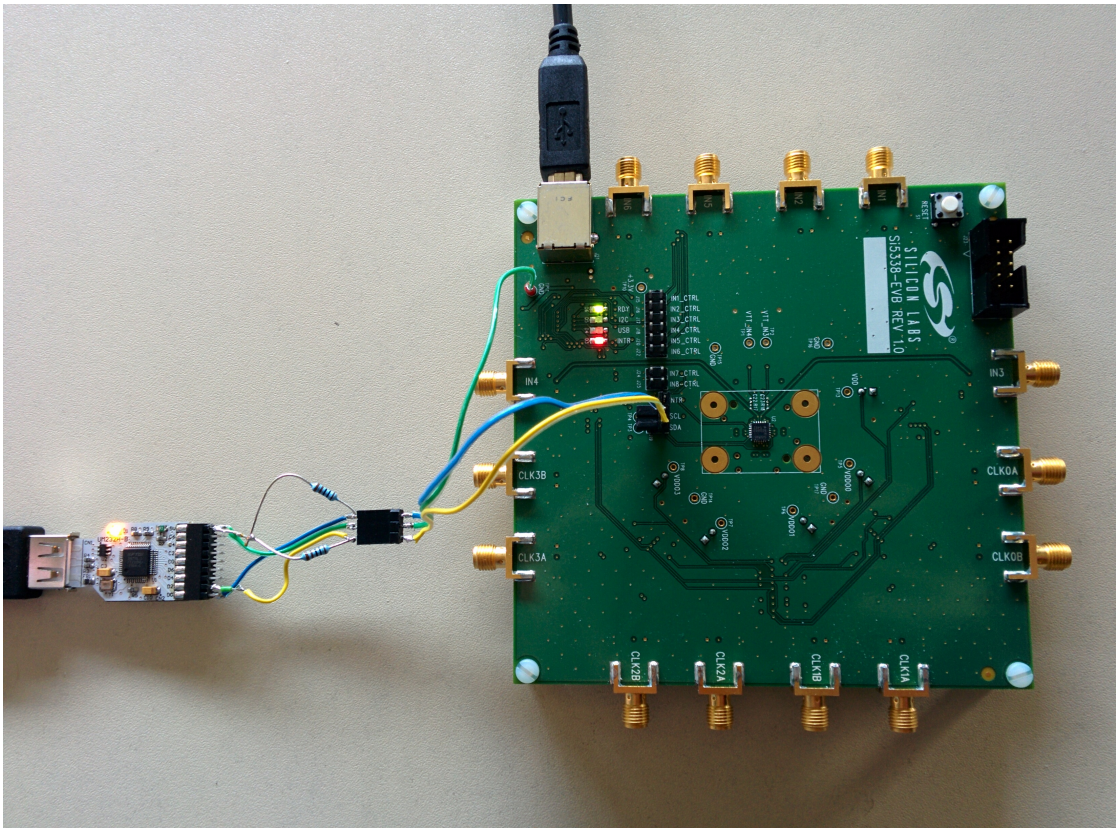


Figure 5.12: Control via External I²C Module [63, 54]

One built-in routine, `--get-part-number`, reads specific registers of the Si5338 device to construct its part number while another one, `--get-status`, reads the status register made up of the four error flags. A part number is constituted of three strings separated by "-". The first one starts with "Si53" followed by a base number of two digits plus a character representing the speed grade. The second one starts with a character representing the revision ID followed by the NVM code of the programmed default configuration. The third one is "GM". An NVM code can be obtained by registering a default configuration at Silicon Labs. As shown in figure 5.13, the application dumps each I²C transaction per line in hexadecimal pairs representing a byte of 8 bits for debugging and confirmation purpose on the screen, followed by the result of the routine prepended by "##". With the default 7-bit slave address of the clock generator of `70` in hexadecimals followed by the direction bit, "0" for writing and "1" for reading, a line begins either with `e0` or `e1`, respectively. That is the slave address shifted by one bit to the left plus the direction bit as least significant bit (LSB), as described in section 3.1.2. The first invocation of routine `--get-part-number` was done while the device was not being connected. Normally, an I²C master module would abort and complain about the slave address not being

acknowledged. The implementation of the external I²C module selected by option `--port` ignores this error. Due to the pull-up resistor of the I²C data line, bits of logical "1" were permanently being read, which resulted in bytes of value `ff`. This is a pseudo part number with base number of "63" and a speed grade of "H" while values of `ff` are out of the defined ranges for revision ID and NVM code but caught by the library `libsi53xx` and substituted with "_" and "?????", respectively. With a connected evaluation board, the part number has correctly been constructed. The device has a base number of "38", is of speed grade "N" with revision ID of "B", and does not have the NVM programmed represented by "00000". To read the register values for the part number, their register addresses must be written to the device firstly. In this case, a continuous sequence of six registers, that is having incremental addresses, was detected and read in burst mode. Thus, only the first register address of the sequence has been written, `e0 00`. Then the six register values have been read by a single I²C transaction, `e1 01 00 26 70 00 00`, dumped on a single line.

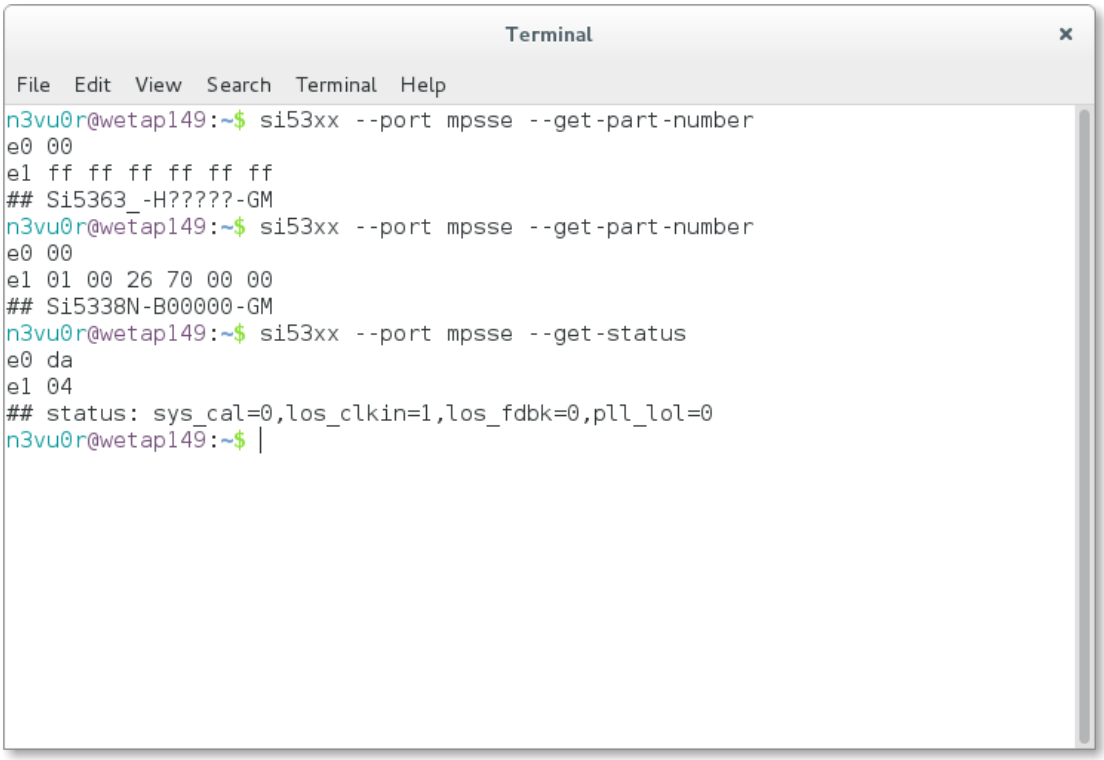


Figure 5.13: Part Number & Status

The invocation of routine `--get-status` reports the four error flags. Since no clock input is given, it reports "los_clkin=1", loss of clock input, as expected. This error flag can be ignored by manipulating the interrupt mask appropriately which

would switch off the red interrupt LED. As shown in figure 5.14, the interrupt status was read, confirming the lighting red LED. Afterwards, the interrupt mask was read. All error flags were set, that means, they were armed for possible triggering the interrupt pin. Now, the flag "los_clkin" was disarmed by invoking routine `--set-interrupt-mask` with "los_clkin=0" as argument. The red interrupt LED was immediately switched off, confirming the last invocation of routine `--get-interrupt` which reported that the interrupt status had been reset.

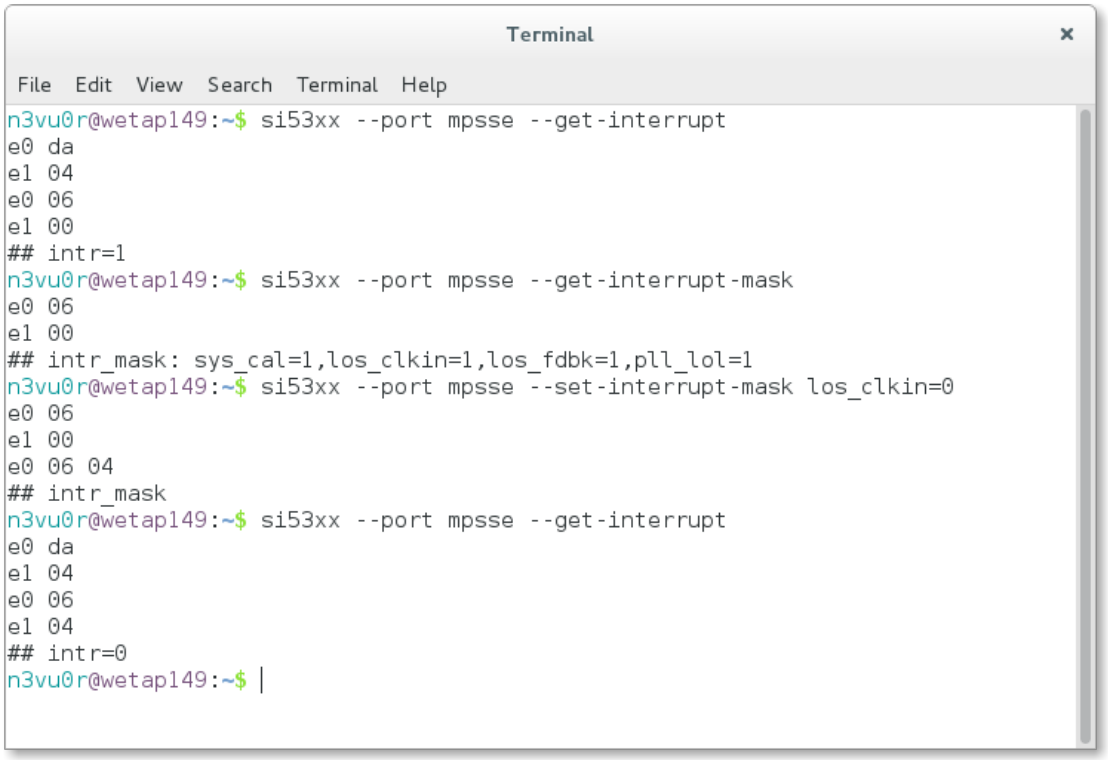


Figure 5.14: Interrupt Mask

This test is a success. It has demonstrated that the built-in routines are functioning as expected. That means, the register descriptions of the Si5338 reference manual [51] along with the single read, single write, and read-modify-write routines are correctly implemented. Detection of register sequences with incremental addresses followed by a burst read has been confirmed as well.

5.4.2 Register & Transition Maps

This setup, shown in figure 5.15, has tested the dynamic control of an output clock derived from an input clock by modifying the device configuration with register map

files. In the meantime, the mezzanine card of section §3.2 was available and the test described in section §5.3 had been repeated to confirm its I²C connectivity. Afterwards, an implementation for the abstract hardware interface of the library `libsi53xx` of section §4.3 was written. The application `si53xx` chooses it as default if not otherwise specified by the option `--port`. This implementation is not specific for the mezzanine card. Instead, it is based on a mainline kernel driver called `i2c-dev` [70] available for other CPU architectures as well. This module can be monitored and configured by a common software package called `i2c-tools` [71], hence the name `i2c2l` was chosen for this implementation with "2l" pronounced "tool".

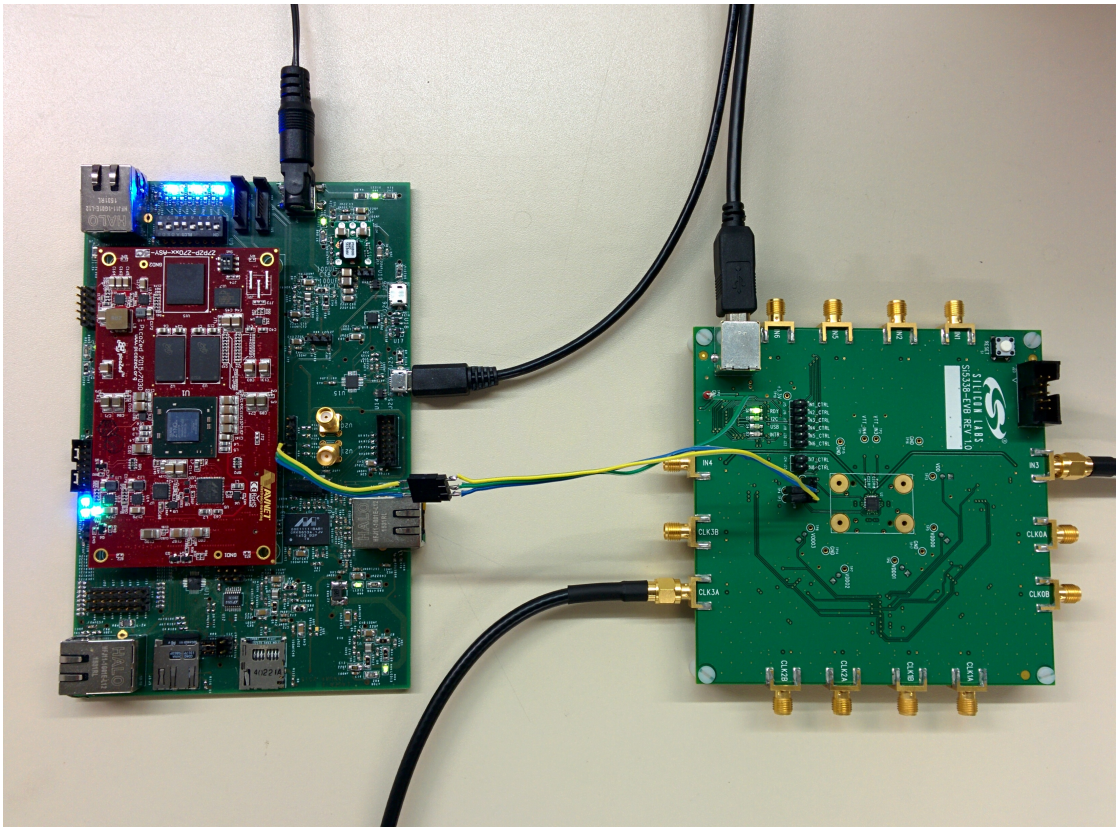


Figure 5.15: Control via Internal I²C Module [17, 54]

By default, the daughter module of the mezzanine card, does not expose its I²C master module being multiplexed with an embedded MultiMediaCard (eMMC) controller. To select it, the positions of two resistor jumpers labeled "JT5" and "JT6", shown at the bottom of figure 5.16, were changed both from position "1-2" to "2-3" as described in the hardware user guide [17] of the PicoZed.

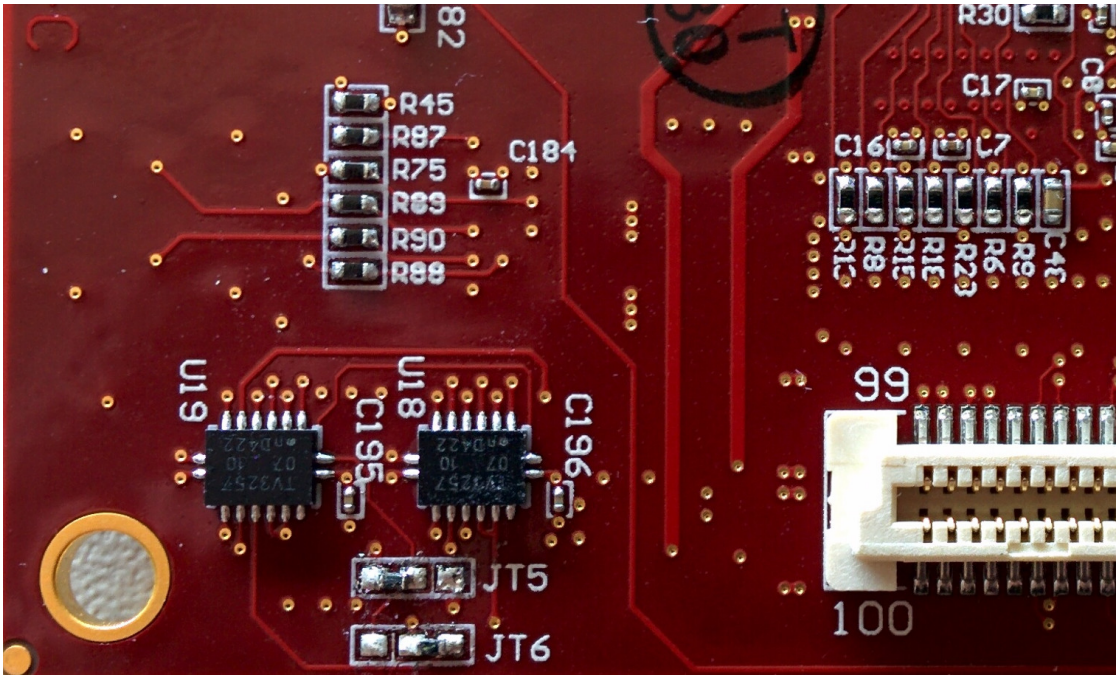


Figure 5.16: Multiplexer Select Exposing Internal I²C Master Module [17]

Two SMA cables of figure 5.15 were used to serve the input clock on connector “IN3” generated by a function generator [72] and to monitor the output clock on connector “CLK3A” with a digital oscilloscope [73] on channel 3. The input clock was actually generated twice to be monitored on channel 1 of the oscilloscope as well. Both single-ended square wave signals of $40.0787\,\mathrm{MHz}$ with peak-to-peak amplitude of $0.6\,\mathrm{V}$ of the input clock were confirmed to be synchronous. The output clock was driven as stub series terminated logic (SSTL) with peak-to-peak amplitude of $1.8\,\mathrm{V}$. Since the output drivers of the function generator and clock generator have resistances of $50\,\Omega$ and the probes of the oscilloscope are terminated with $50\,\Omega$ as well, the amplitudes of the signals are expected to be halved. The configuration of the clock generator was created with the help of the ClockBuilder Desktop Software and saved as C code header file. Afterwards, it was converted with the script `si53xx-map` resulting in a file named “default.map”. It generates an output clock of identical frequency of the input clock. The application `si53xx` finally wrote the register map to the clock generator, as shown in figure 5.17. The first twenty lines are dumped to the screen. The line `e0 1c 0b 49 b0 e3 e3 e3 c0 80` is a burst write of a sequence of eight registers with the first register address of `1c`.

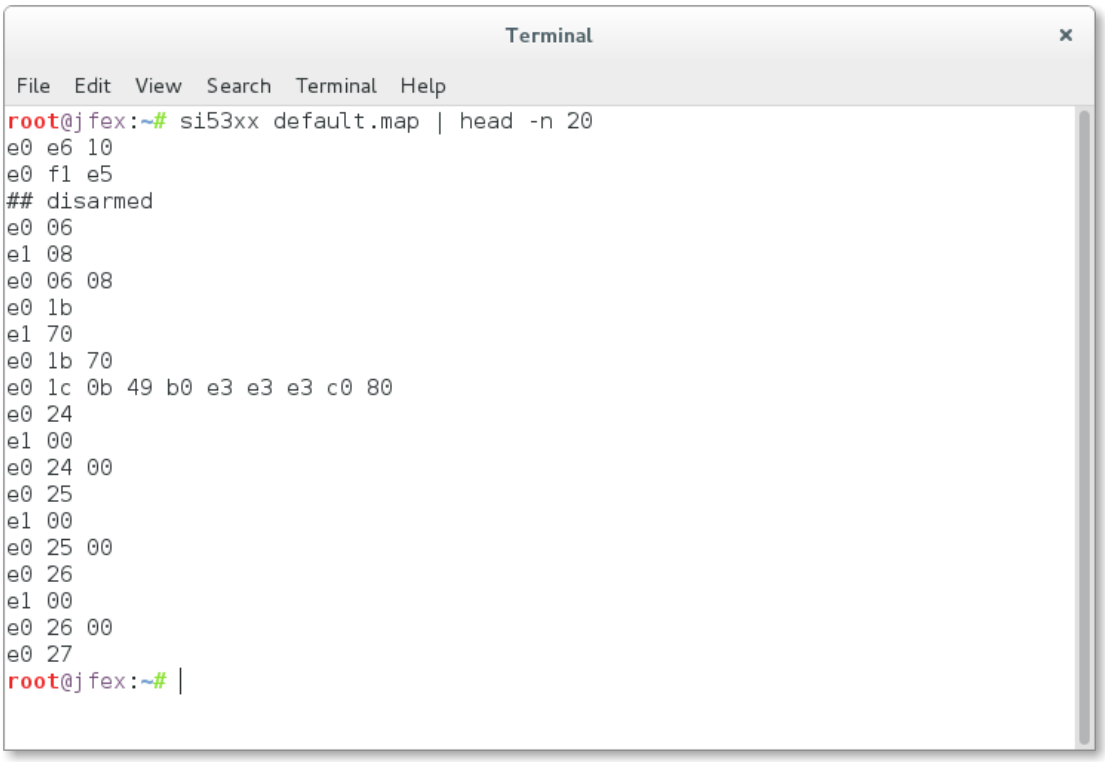


Figure 5.17: Default Register Map

In figure 5.18, the output clock signal generated by the clock generator is drawn blue while the input clock signal is drawn yellow. The oscilloscope is capable of measuring frequencies. For the input clock a mean frequency of roughly $40.0785\,\mathrm{MHz}$ and for the output clock a mean frequency of roughly $40.0792\,\mathrm{MHz}$ are measured. Due to possible jitter, a short time span to calculate the mean values, and a limited accuracy of the clock generator and the oscilloscope, there are slight deviations from the supposed frequency of $40.0787\,\mathrm{MHz}$. But it confirms that the default configuration has correctly been written to the clock generator and that a burst write has correctly been detected and performed.

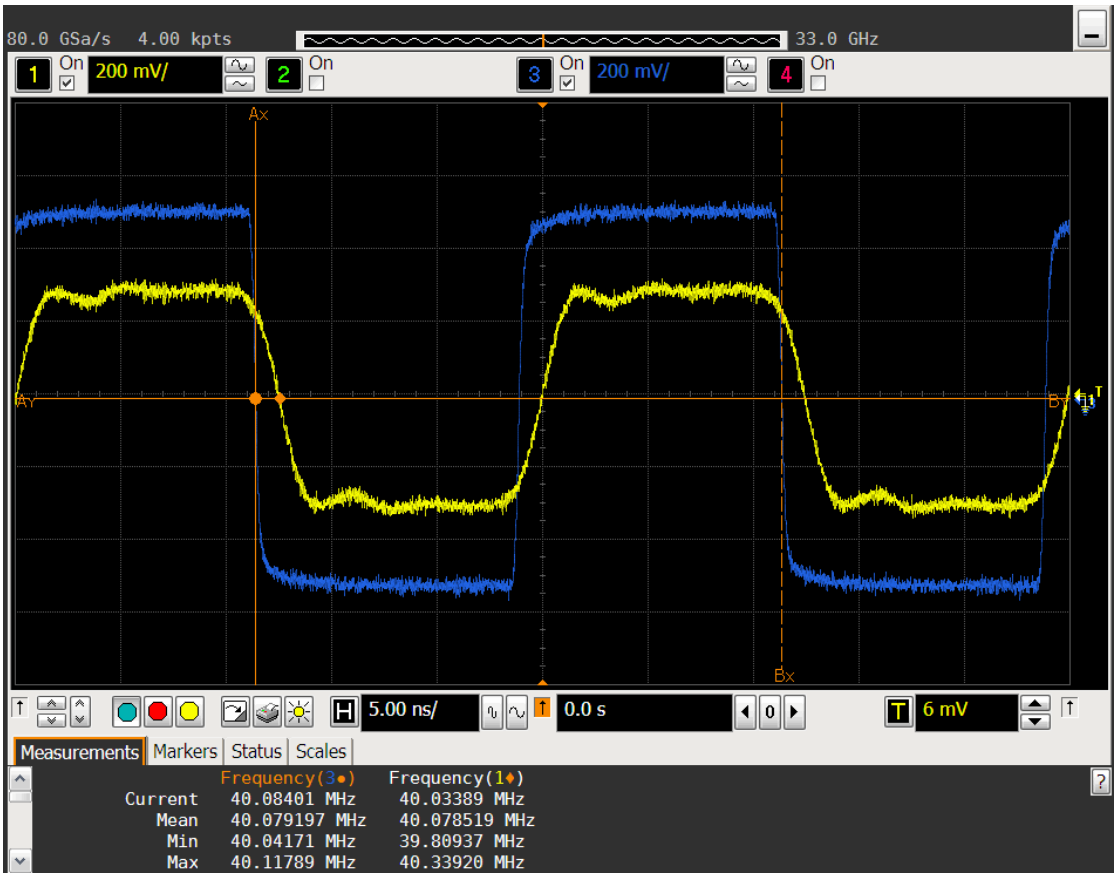


Figure 5.18: Default Clock Frequency [73]

Now, the output clock frequency was supposed to be doubled to 80.1574 MHz. The same steps were done to create an appropriate register map file but this time it was named “clk3@80.map”. Instead of writing the whole register map file, a pair of transition map files was created with the script `si53xx-cmp`. It generated two files, “clk3@80.pos” and “clk3@80.neg”, containing only the differences of the register map files “default.map” and “clk3@80.map”. Former contains the registers necessary to be written to switch to the doubled clock frequency while latter contains the registers necessary to be written to switch back to the original clock frequency. The application `si53xx` was invoked with the transition map file “clk3@80.pos”, as shown in figure 5.19. All registers, written to the device, are dumped on the screen. In general, before register maps are being written, the device is disarmed. That means, its outputs are disabled and the LOL state machine is paused. After one or more register maps have been written, the device is armed again. That means, the input clocks are validated, the PLL is locked to the input clock with its new configuration, the voltage-controlled oscillator (VCO) is recalibrated, the LOL state machine is resumed, and the outputs are enabled again.

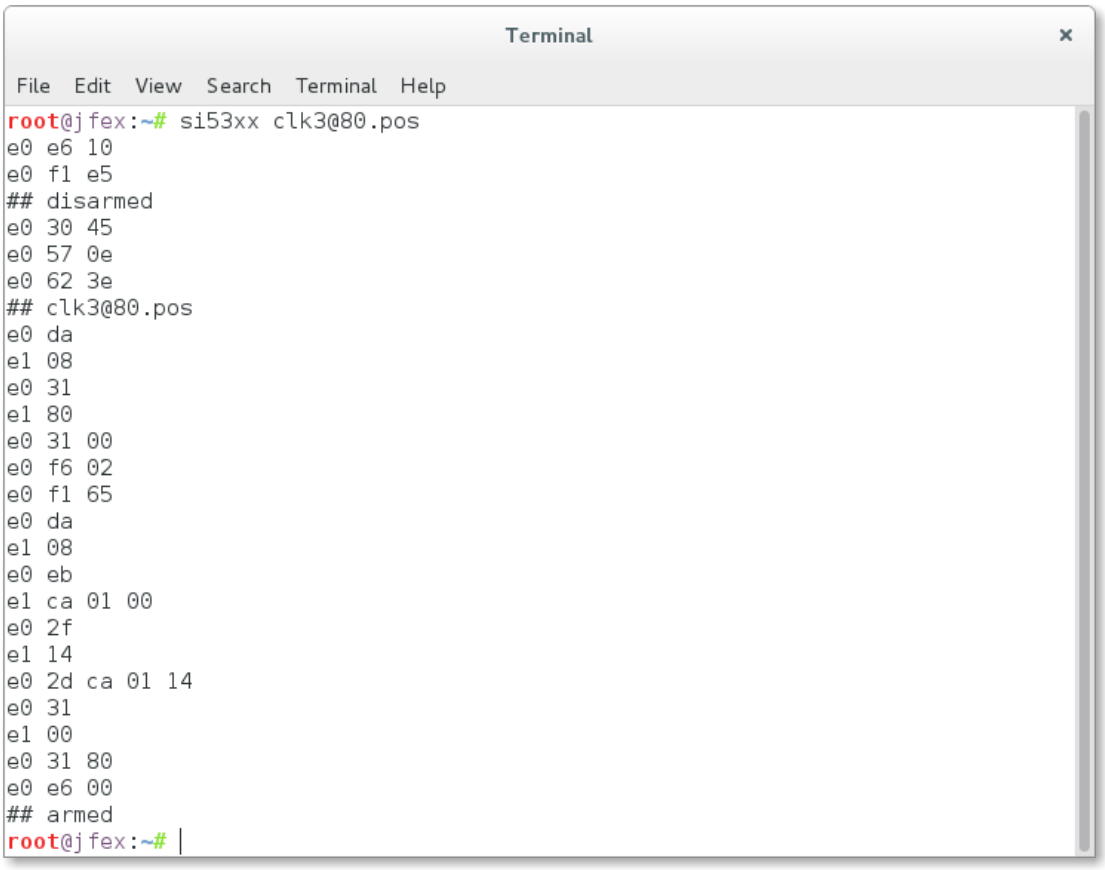


Figure 5.19: Positive Transition Map

In this case the transition map contains only three I²C transactions. The resulting output clock is drawn, again in blue, in figure 5.20. Its frequency measuring reported roughly 80.1595 MHz. Apart from a slight deviation, this is the expected doubled clock frequency. Thus, the positive transition map file has correctly been generated and written to the device. The implementations of the arm and disarm routines have successfully recalibrated the device.

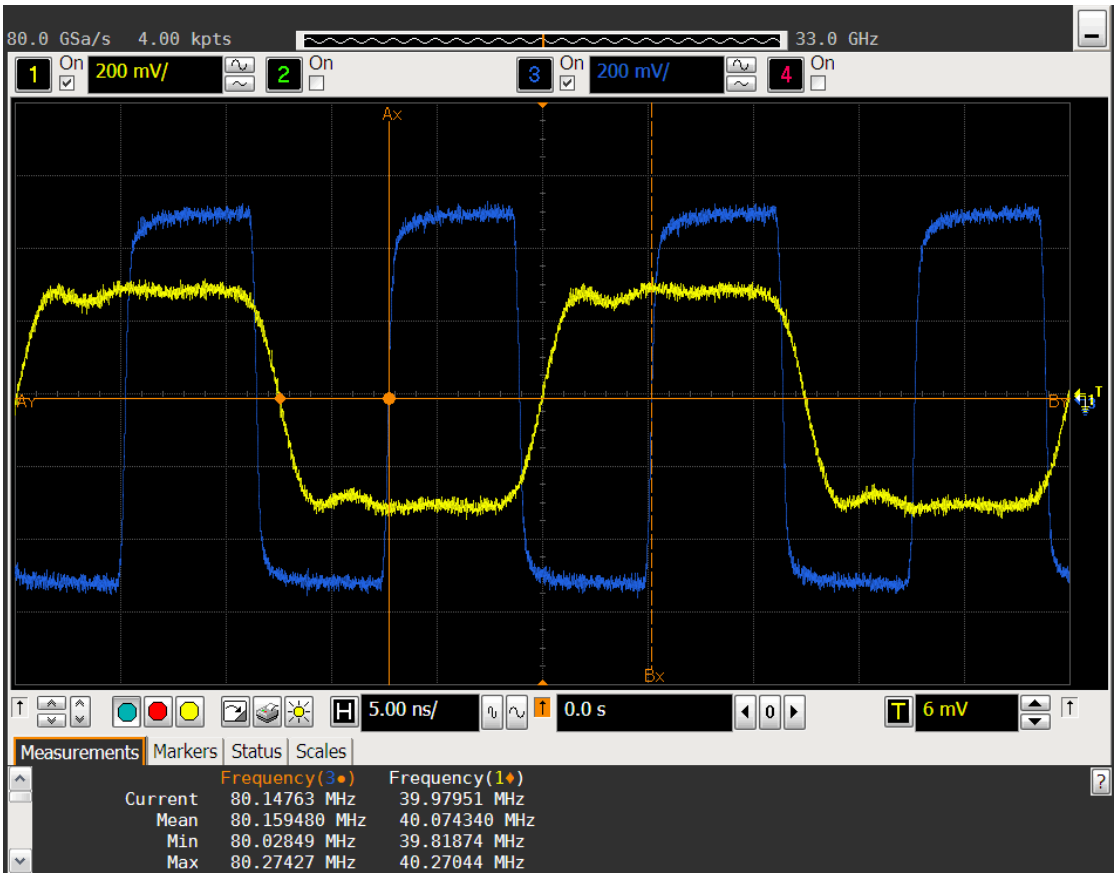


Figure 5.20: Doubled Clock Frequency [73]

Finally, the negative transition map file was written, as shown in figure 5.21. Again, only three I²C transactions are required. The same registers of addresses `30`, `57`, and `62` have been written. But this time with different values compared with the positive transition map file. The output clock frequency has correctly been restored to the original one.

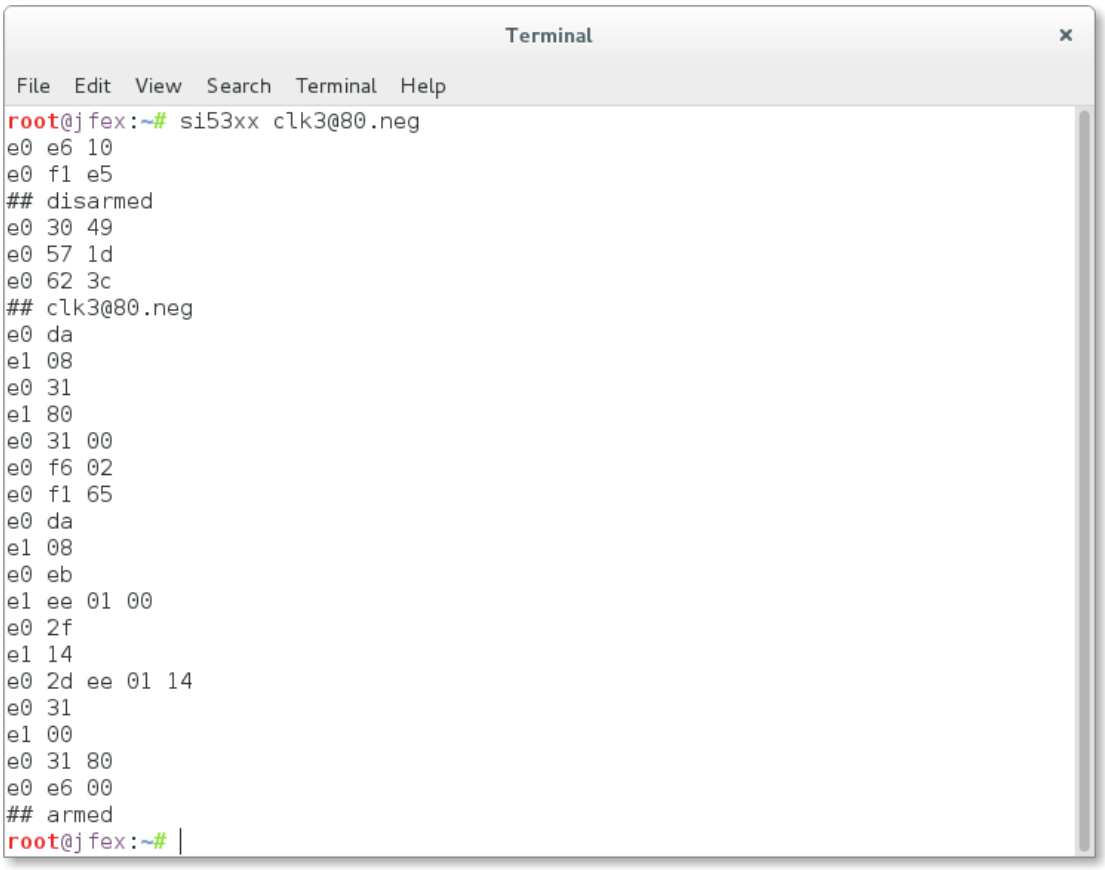


Figure 5.21: Negative Transition Map

This test has successfully demonstrated the functionality of controlling the clock generator by the application `si53xx`. It was possible to write register as well as transition maps. The conversion and comparison scripts, `si53xx-map` and `si53xx-cmp`, successfully complement the control application `si53xx`.

Chapter 6

Conclusion & Outlook

The designed mezzanine card of section §3.3 is capable of booting into a terminal login prompt. Thus, the schematics correctly interface its core components. All modifications to the boot image and the root file system of the operating system are successfully managed with the workflow kit of section §4.1, allowing reproducible image generation required for long-term use and maintenance. Two mistakes of the hardware design have been made. The SD card slots have been wrongly oriented, not pointing away from the mezzanine card. But since no high components are close to them, SD cards can still be plugged into both slots. Another drawback is the lack of a button triggering a soft reset of the Zynq chip. Instead, when debugging the stopping of the boot process, hard resets were being made by unplugging and replugging the external power supply.

The hybrid approach, that is having both a CPU and an FPGA inside a single chip, gives valuable advantages by having an operating system running on the CPU with a whole ecosystem of applications and libraries. This allows the developed software packages to reuse general-purpose functionalities implemented by proven libraries. Furthermore, it allows the bit stream files for programming the processor FPGAs to be stored on a local SD card managed by common tools of the operating system. Communication channels between CPU and FPGA have successfully been established with the file transfer application of section §4.2. The achieved throughput rates with a maximum write rate of $(9.737 \pm 0.019)\,\mathrm{MB/s}$ fulfill the requirement of programming all four processor FPGAs in a reasonable amount of time. The jFEX can be programmed within less than a minute, $(32.86 \pm 0.07)\mathrm{s}$ to be precisely. Remote updates of bit stream files and software applications are transferred over Ethernet via CPU, managed by the workflow kit as well. This facilitates maintenance and development processes by not unplugging and replugging the SD card for every single

modification. Among these advantages of TCP based connectivity done by the CPU, the hybrid approach simultaneously allows the required UDP based IPBus communication for controlling the FPGA. As a precautionary measure regarding possible future changes to the central control interface of section 3.1.1, TCP based control is possible as well by connecting the Ethernet cable from the Ethernet jack of the backplane to the Ethernet jack of the CPU instead of the one of the FPGA. In this case, the Ethernet jack of the CPU would not only be used as a separate maintenance and development interface. Instead, it would additionally serve as control interface but accessed through the backplane. Since software development was focused on usability and extensibility, implementations like the CPU/FPGA communication have been externalized into libraries. This would allow TCP based control communication to be easily routed from CPU to FPGA with the same library used by the file transfer application.

The I²C connectivity of the mezzanine card has successfully been used to control the clock generator with the software package of section §4.3. Due to the fact that software has been structured from abstract interfaces down to concrete implementations, it was possible to interface the clock generator via internal as well as external I²C modules with little programming effort. The external I²C module served as initial test device for general I²C communication as well as alternative to the, at that time not yet available, mezzanine card for debugging the software package. The two scripts of this software package allow the management of whole device configurations and the generation of transition maps from one configuration to another. They successfully complement the control application by individually extracting a single feature of interest from a whole device configuration in order to enable or disable it.

The next steps are to confirm the correctness of the schematics interfacing the PHY of the mezzanine card for Ethernet communication via FPGA and to control further components of the jFEX like its power modules. The former step is required before producing the second iteration of the mezzanine card. That iteration will be pluggable onto the jFEX since in the meantime the pin layout of its header has been finalized.

Bibliography

[1] The ATLAS Collaboration: The ATLAS Experiment at the CERN Large Hadron Collider (2008 JINST 3 S08003 – August 14, 2008)
https://cdsweb.cern.ch/record/1129811/files/jinst8_08_s08003.pdf

[2] CERN: The Large Hadron Collider – Facts (CERN Brochure – 2009)
https://cds.cern.ch/record/1165534/files/CERN-Brochure-2009-003-Eng.pdf

[3] Matilda Heron – CERN: ATLAS and CMS experiments shed light on Higgs properties (2015-09-01)
https://cds.cern.ch/record/2059194

[4] The ATLAS Collaboration: Technical Design Report for the Phase-I Upgrade of the ATLAS TDAQ System (CERN-LHCC-2013-018, ATLAS-TDR-023 – 30 November 2013)
https://cds.cern.ch/record/1602235/files/ATLAS-TDR-023.pdf

[5] Ivana Hristova, The ATLAS Collaboration: The Phase-I Upgrade of the ATLAS First Level Calorimeter Trigger (L1Calo) – Technology and Instrumentation in Particle Physics (TIPP) – International Conference (2-6 June 2014, Amsterdam)
https://indico.cern.ch/event/192695/contributions/353430/attachments/277277/387894/TIPP14_hristova.pdf

[6] Sebastian Artz, Stefan Rave, Ulrich Schäfer, Esteban Torregrosa: Technical Specification – ATLAS Level-1 Calorimeter Trigger Upgrade – Jet Feature Extractor (jFEX) Prototype (Draft, Version: 0.3 – ~ October 2014)
http://www.staff.uni-mainz.de/rave/jFEX_PDR/jFEX_spec_v0.3.pdf

[7] Dan Noyes – CERN: About CERN (2012-01-19)
https://cds.cern.ch/record/1997225

[8] Ian Brawn: L1Calo Phase I Status – eFEX, jFEX & gFEX, Hub & ROD, TREX, Link-Speed Tests, Firmware Management, Schedule (TDAQ Week – 25 May 2016)
https://indico.cern.ch/event/464852/contributions/2177460/attachments/1278755/1898686/L1Calo_160525_TDAQ.pdf

[9] Xilinx, Inc.: UltraScale Architecture GTH Transceivers – User Guide (UG576, v1.3 – November 24, 2015)
http://www.xilinx.com/support/documentation/user_guides/ug576-ultrascale-gth-transceivers.pdf

[10] Xilinx, Inc.: UltraScale Architecture Configuration – User Guide (UG570, v1.6 – December 16, 2015)
http://www.xilinx.com/support/documentation/user_guides/ug570-ultrascale-configuration.pdf

[11] NXP Semiconductors N.V.: I²C-bus specification and user manual (UM10204, Rev. 6 – 4 April 2014)
https://www.nxp.com/documents/user_manual/UM10204.pdf

[12] The Internet Engineering Task Force: Transmission Control Protocol (RFC 793 – September 1981)
https://www.ietf.org/rfc/rfc793.txt

[13] The Internet Engineering Task Force: User Datagram Protocol (RFC 768 – 28 August 1980)
https://www.ietf.org/rfc/rfc768.txt

[14] Robert Frazier, Greg Iles, Dave Newbold, Andrew Rose: The IPbus Protocol & The IPbus Suite (Talk: TIPP 2011 – 09/06/11)
https://indico.cern.ch/event/102998/contributions/17149/attachments/10532/15402/TIPP_Frazier_09_06_2011.pdf

[15] Linear Technology Corporation: RS-232 Single and Dual Transceivers – LTC2801/LTC2802/LTC2803/LTC2804 (2801234fe)
https://cds.linear.com/docs/en/datasheet/2801234fe.pdf

[16] Silicon Laboratories, Inc. (Silicon Labs): Single-Chip USB-to-UART Bridge (CP2104, Rev. 1.1 – 11/2013)
https://www.silabs.com/Support%20Documents/TechnicalDocs/cp2104.pdf

[17] Avnet, Inc.: PicoZed Z7015 / Z7030 SOM (System-On Module) – Hardware User Guide (Version 1.5 – 3/11/16)

http://zedboard.org/sites/default/files/documentations/PicoZed%207015_7030_User%27s_Guide_v1.5.pdf

[18] Xilinx, Inc.: Zynq-7000 All Programmable SoC Overview – Product Specification (DS190, v1.9 – January 20, 2016)
http://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf

[19] Avnet, Inc.: MicroZed Zynq™ Evaluation and Development and System on Module – Hardware User Guide (Version 1.6 – 22 January 2015)
http://zedboard.org/sites/default/files/documentations/MicroZed_HW_UG_v1_6.pdf

[20] Texas Instruments Inc. (TI): TPS82085 – 3-A High Efficiency Step-Down Converter MicroSiP™ with Integrated Inductor (SLVSCN4B – October 2014 – Revised August 2015)
https://www.ti.com/lit/ds/symlink/tps82085.pdf

[21] Xilinx, Inc.: AR# 47817 – Design Advisory for the Kintex-7 and Virtex-7 GTX Transceiver Power-up/Power-down (02/18/2013)
http://www.xilinx.com/support/answers/47817.html

[22] Texas Instruments Inc. (TI): TXS02612 – SDIO Port Expander with Voltage-Level Translation (SCES682C – December 2008 – Revised February 2009)
https://www.ti.com/lit/ds/symlink/txs02612.pdf

[23] Xilinx, Inc.: AR# 43989 – 7 Series FPGAs - LVDS_33, LVDS_25, LVDS_18, LVDS inputs & outputs for High Range (HR) and High Performance (HP) I/O banks (10/14/2014)
http://www.xilinx.com/support/answers/43989.html

[24] Serial ATA International Organization (SATA-IO): Technical Overview
https://www.sata-io.org/technical-overview

[25] Institute of Electrical and Electronics Engineers (IEEE): 1149.1-2013 – IEEE Standard for Test Access Port and Boundary-Scan Architecture
https://standards.ieee.org/findstds/standard/1149.1-2013.html

[26] David MacKenzie, Tom Tromey, Alexandre Duret-Lutz, Ralf Wildenhues, Stefano Lattarini: GNU Automake – Manual (For Version 1.15 – 31 December 2014)
https://www.gnu.org/software/automake/manual/automake.pdf

[27] Debian Project: Debian “jessie” Release Information
https://www.debian.org/releases/jessie/

[28] Xilinx, Inc.: PetaLinux Tools (2015.4 – Dec 15, 2015)
http://www.xilinx.com/petalinux

[29] Xilinx, Inc.: Vivado Design Suite (2015.4 – Nov 18, 2015)
http://www.xilinx.com/vivado

[30] John Ousterhout, Tcl Core Team: Tool Command Language (TCL)
https://www.tcl.tk/

[31] Richard M. Stallman, Roland McGrath, Paul D. Smith: GNU Make – Manual (Version 4.1 – September 2014)
https://www.gnu.org/software/make/manual/make.pdf

[32] Xilinx, Inc.: Build FSBL
http://www.wiki.xilinx.com/Build+FSBL

[33] DENX Software Engineering: Das U-Boot - The Universal Boot Loader
http://www.denx.de/wiki/U-Boot

[34] Linaro Engineering Organization: The Devicetree Specification
http://www.devicetree.org/

[35] Linux Kernel Organization, Inc.: The Linux Kernel Archives
https://www.kernel.org/

[36] J. W. Hunt – Department of Electrical Engineering - Stanford University, M. D. McIlroy – Bell Laboratories: An Algorithm for Differential File Comparison
http://www.cs.dartmouth.edu/~doug/diff.pdf

[37] Chet Ramey – Case Western Reserve University, Brian Fox – Free Software Foundation: Bash Reference Manual (For Version 4.3 – February 2014)
https://www.gnu.org/software/bash/manual/bash.pdf

[38] Andrew Tridgell, Paul Mackerras – Department of Computer Science - Australian National University: The rsync algorithm (1998-11-09)
https://rsync.samba.org/tech_report/tech_report.html

[39] OpenBSD Project: OpenSSH
http://www.openssh.com/

[40] Mel Gorman – Linux Kernel Organization, Inc.: Understanding The Linux Virtual Memory Manager (July 9, 2007)
https://www.kernel.org/doc/gorman/pdf/understand.pdf

[41] Andries Brouwer, Michael Kerrisk: Linux Programmer's Manual – MMAP (2016-03-15)
http://man7.org/linux/man-pages/man2/mmap.2.html

[42] David S. Miller, Richard Henderson, Jakub Jelinek: Dynamic DMA mapping Guide
https://www.kernel.org/doc/Documentation/DMA-API-HOWTO.txt

[43] Xilinx, Inc.: AXI GPIO v2.0 – LogiCORE IP Product Guide – Vivado Design Suite (PG144 – November 18, 2015)
http://www.xilinx.com/support/documentation/ip_documentation/axi_gpio/v2_0/pg144-axi-gpio.pdf

[44] Arseny Kapoulkine: pugixml – Light-weight, simple and fast XML parser for C++ with XPath support (1.7 release – 19 October 2015)
http://pugixml.org/

[45] World Wide Web Consortium (W3C): Extensible Markup Language (XML) 1.0 (Fifth Edition) – W3C Recommendation (26 November 2008)
https://www.w3.org/TR/2008/REC-xml-20081126/

[46] C++ Reference
http://en.cppreference.com/w/cpp

[47] Richard M. Stallman and the GCC Developer Community: Using the GNU Compiler Collection – 6.42 An Inline Function is As Fast As a Macro (For Version 6.1.0)
https://gcc.gnu.org/onlinedocs/gcc-6.1.0/gcc.pdf

[48] Richard M. Stallman and the GCC Developer Community: Using the GNU Compiler Collection – 6.44.2 Extended Asm - Assembler Instructions with C Expression Operands (For Version 6.1.0)
https://gcc.gnu.org/onlinedocs/gcc-6.1.0/gcc.pdf

[49] Institute of Electrical and Electronics Engineers (IEEE): 1076-2008 – IEEE Standard VHDL Language Reference Manual
https://standards.ieee.org/findstds/standard/1076-2008.html

[50] Silicon Laboratories, Inc. (Silicon Labs): Si5338 – I²C-Programmable Any-Frequency, Any-Output Quad Clock Generator (Si5338, Rev. 1.6 – 12/2015)
https://www.silabs.com/Support%20Documents/TechnicalDocs/Si5338.pdf

[51] Silicon Laboratories, Inc. (Silicon Labs): Si5338 Reference Manual – Configuring the Si5338 Without ClockBuilder Desktop (Si5338-RM, Rev. 1.3 – 8/2014)
https://www.silabs.com/Support%20Documents/TechnicalDocs/Si5338-RM.pdf

[52] Silicon Laboratories, Inc. (Silicon Labs): Jump Start: In-System, Flash-Based Programming for Silicon Labs' Timing Products (AN428, Rev. 0.6 – 10/2010)
https://www.silabs.com/Support%20Documents/TechnicalDocs/AN428.pdf

[53] Silicon Laboratories, Inc. (Silicon Labs): Si5338/35/34/56 ClockBuilder Desktop Software (Version 6.4 – October 8, 2014)
http://www.silabs.com/Support%20Documents/Software/ClockBuilderDesktopSwInstall.zip

[54] Silicon Laboratories, Inc. (Silicon Labs): Si5330/34/35/38 Evaluation Board User's Guide (Si5338-EVB, Rev. 1.4 – 11/2011)
https://www.silabs.com/Support%20Documents/TechnicalDocs/Si5338-EVB.pdf

[55] Silicon Laboratories, Inc. (Silicon Labs): Si5338/Si5356 Field Programmer Kit User's Guide (Si5338/56-PROG-EVB, Rev. 0.4 – 06/2010)
https://www.silabs.com/Support%20Documents/TechnicalDocs/Si5338-56-PROG-EVB.pdf

[56] Silicon Laboratories, Inc. (Silicon Labs): Si5356 – I²C-Programmable, Any-Frequency 1-200 MHz, Quad Frequency 8-Output Clock Generator (Si5356A, Rev. 1.3 – 2014)
https://www.silabs.com/Support%20Documents/TechnicalDocs/Si5356A.pdf

[57] C Reference
http://en.cppreference.com/w/c

[58] Free Software Foundation, Inc.: sed, a stream editor (For Version 4.2.1 – August 24, 2010)
https://www.gnu.org/software/sed/manual/sed.html

[59] Richard M. Stallman and the GCC Developer Community: Using the GNU Compiler Collection (For Version 4.9.3)
https://gcc.gnu.org/onlinedocs/gcc-4.9.3/gcc.pdf

[60] Xilinx, Inc.: Zynq-7000 All Programmable SoC – Technical Reference Manual (UG585, v1.10 – February 23, 2015)
http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf

[61] SD Association: Bus Speed (Default Speed / High Speed / UHS)
https://www.sdcard.org/developers/overview/bus_speed/index.html

[62] Xilinx, Inc.: Virtex UltraScale FPGAs Data Sheet – DC and AC Switching Characteristics (DS893, v1.7.1 – April 4, 2016)
http://www.xilinx.com/support/documentation/data_sheets/ds893-virtex-ultrascale-data-sheet.pdf

[63] Future Technology Devices International Ltd. (FTDI): UM232H-B USB to Serial/Parallel Break-Out Module (UM232H-B, 1.2)
http://www.ftdichip.com/Support/Documents/DataSheets/Modules/DS_UM232H-B.pdf

[64] libmpsse – MPSSE Library (v1.3)
https://github.com/devttys0/libmpsse

[65] Intra2net AG: libftdi – FTDI USB driver with bitbang mode (Version 1.3 – 2016-05-20)
https://www.intra2net.com/en/developer/libftdi/

[66] libusb – C library that gives applications easy access to USB devices
http://libusb.info/

[67] RN-Control – Universal AVR Controller Board
http://rn-wissen.de/wiki/index.php?title=RN-Control

[68] Atmel Corporation: ATmega32, ATmega32L – 8-bit AVR Microcontroller with 32KBytes In-System Programmable Flash (2503Q–AVR–02/11)
http://www.atmel.com/images/doc2503.pdf

[69] Thomas Fischl: USBasp – USB programmer for Atmel AVR controllers (2011-05-28)
http://www.fischl.de/usbasp/

[70] Simon G. Vogl, Frodo Looijaard: I²C Bus – Character Device Interface (v1.9 – 2001/08/15)
https://www.kernel.org/pub/linux/kernel/people/marcelo/linux-2.4/include/linux/i2c-dev.h

[71] Debian Project: Heterogeneous set of I²C tools for Linux (i2c-tools, 3.1.1-1)
https://packages.debian.org/jessie/i2c-tools

[72] Tektronix, Inc.: AFG3000 Series of Function, Arbitrary Waveform, and Pulse Generators – AFG 3011 / 3021B / 3022B / 3101 / 3102 / 3251 / 3252 Datasheet
http://www.tek.com/sites/tek.com/files/media/media/resources/AFG3000_Series_Arbitrary-Function_Generators_Datasheet_76W-18656-5.pdf

[73] Agilent Technologies: Agilent Infiniium 90000 X-Series Oscilloscopes
http://cp.literature.agilent.com/litweb/pdf/5990-5271EN.pdf