A Data Injector for the High Luminosity LHC ATLAS Liquid Argon Signal Processor

by

Maheyer Jamshed Shroff
BASc., University of Toronto, 2018

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Physics and Astronomy

A Data Injector for the High Luminosity LHC ATLAS Liquid Argon Signal Processor

by

Maheyer Jamshed Shroff
BASc., University of Toronto, 2018

Supervisory Committee

Dr. R. McPherson, Co-Supervisor
(Department of Physics and Astronomy)

Dr. R. Keeler, Co-Supervisor
(Department of Physics and Astronomy)

Dr. C. Bohne, Committee Member
(Department of Chemistry)

Supervisory Committee

Dr. R. McPherson, Co-Supervisor
(Department of Physics and Astronomy)

Dr. R. Keeler, Co-Supervisor
(Department of Physics and Astronomy)

Dr. C. Bohne, Committee Member
(Department of Chemistry)

ABSTRACT

A test-bench is created that injects digital pulses that emulate ATLAS LAr Front End Board electronic signal pulses in order to test prototypes. The prototypes are for new electronics for an upgrade to the CERN Large Hadron Collider that increases the rate of proton-proton collisions by an order of magnitude. This High-Luminosity Large Hadron Collider requires a completely new Trigger and Data Acquisition system to deal with information from detectors.

One such system that is currently being developed is the Liquid Argon Signal Processor (LASP) whose architecture is based on Field Programmable Gate Arrays (FPGA). Validation of individual modules of the LASP are of key importance in the development cycle. Additionally, verification of module behaviour with real ATLAS pulses will not be available until much later in the project timeline.

The injector project is implemented on an Intel Stratix 10 FPGA, using a soft-core NIOS II processor for TCP/IP communication with a workstation in order to transfer Monte Carlo simulation pulses to the FPGA, where it is then stored in a 2 GB DDR3 external memory. The pulses are then retrieved into internal memory buffers and are transmitted to the LASP at 40 MHz. The user is in complete control of the data pulses injected which is a vital property that would test LASP behaviour for different cases and possible failure modes.

Contents

List of Tables

List of Figures

Glossary of Terms, Abbreviations and Acronyms.

A list of **boldfaced** terms in the text are defined here. An emphasis is given on electronic terms.

ADC	*Analogue to Digital Converter*	An analogue signal that has been converted to a digital signal
ALM	*Adaptive logic modules*	A powerful basic building block in FPGAs that can perform multiple logic operations
API	*Application programming interface*	An interface used by computer programs to request services from the operating system
AREUS	*ATLAS Readout Electronics Upgrade Simulation*	A Monte-carlo program that can be used to simulate LAr signal pulses at the HL-LHC
ARM	*Advanced RISC Machine*	Family of Reduced Instruction set computing (See **RISC**) architectures for processors
ASIC	*Application-specific integrated circuit*	An integrated circuit chip that has been programmed for a specific use only
ATLAS	*A Toridial LHC Apparatus*	A general purpose experiment built around the LHC ring
Avalon Interface		A communications Interface internal to Intel FPGAs
Avalon MM	*Avalon Memory Mapped*	A subset of the Avalon Interface that allows communication with entities that are have a fixed memory address
BC	*Bunch Crossing*	The event when the two oppositely circulating proton beams collide in the LHC
BCID	*Bunch crossing Identification*	A value that is tacked on the signal to indicate which bunch-crossing the signal is from
BCR	*Bunch counter Reset*	A signal that resets the BCID counter
CERN	*Conseil European pour la Recherche Nucleaire*	European research organization that operates the LHC

Checksum		A small-sized datum derived from a block of digital data for the purpose of detecting errors that may have been introduced during its transmission
CMOS	*Complementary Metal Oxide Semiconductor*	A type of semiconductor used in constructing integrated circuits chips
DAQ	*Data acquisition*	Storage system of data that has passed the High Level Trigger
DCFIFO	*Dual Clock FIFO*	A First in First out buffer that has different Input and Output clocks
DDR	*Double Data Rate*	A Random Access Memory device with a high bandwidth
DHCP	*Dynamic Host Configuration Protocol*	An Internet Protocol where a server dynamically assigns an IP address to each device on the network
DMA	*Direct Memory Access*	A feature that allows access to the main memory system independent of the CPU. Implemented in FPGAs as an IP core
DSP	*Digital Signal Processors*	Specialized microprocessors that performs dedicated digital signal processing such as filtering
ECAL	*Electromagnetic Calorimeter*	Measures the energy deposited by photons and electrons (and partly hadrons)
EMB	*Electromagnetic Barrel*	Part of the ECAL, located in-between the two endcaps
EMEC	*Electromagnetic End-Caps*	Part of the ECAL, located at the two ends of the Calorimeter
EMIF	*External Memory Interface*	An Intel IP that provides an interface for the FPGA to communicate with an External Memory Device such as a DDR3
Ethernet		A fast communications protocol that can reach speeds of 10Gb/s in specialized FPGAs

FEB	*Front End Board*	Part of the on detector electronics which receive signals from the calorimeter. Upgraded FEBs will be called FEB2s
FELIX	*Front End Link eXchange*	Network interface module that is compatible with the lpGBT protocol
FEX	*Feature Exchange*	Receives processed energy measurements from the LASP (or LDPS in Phase I)
FF	*Flip Flop*	A simple circuit that has two stable states and is used to store information
FIFO	*First in First out*	Data buffer where the first entry in a queue is processed first
FPGA	*Field Programmable Gate Arrays*	Flexible Integrated circuits containing programmable switches and configurable logic cells
GMII	*Gigabit Media Independent Interface*	Interface used to connect an Ethernet MAC layer to an Ethernet PHY layer
HAL	*Hardware Abstraction Layer*	A software that provides a set of routines that can access hardware resources
HDL	*Hardware Description Language*	A specialized language used to describe the structure of digital logic circuits
HEC	*Hadronic Endcap Calorimeter*	Measures the energy deposited by hadrons at the backward end of the calorimeter
HL-LHC	*High Luminosity Large Hadron Collider*	The upgraded version of the LHC which will operate at 10 times the number of current LHC collisions
HLT	*High Level Trigger*	Software based trigger that uses hit identification to reduce the event rate to interesting physics events
I/O	*Input/Output*	Pieces of Hardware or an interface that provides Input and Output operations
Integrated Luminosity, L		Integral of luminosity with respect to time which corresponds to the amount of data obtained

IP	*Intellectual Property*	Set of HDL cores that solve particular common problems and are provided by third-party sources
IP	*Internet Protocol*	A communications protocol describing how to relay data across network boundaries
JTAG-UART	*Joint Test Action Group-Universally Asynchronous Receiver-Transmitter*	An interface through which FPGAs can communicate typically with a computer
LAB	*Logic Array Block*	A group of 10 ALMs
LAN	*Local Area Network*	A computer network that interconnects computers within a limited area
LAr	*Liquid Argon*	The active medium of the calorimeter in ATLAS. Sometimes loosely referencing the calorimeter itself
LASP	*Liquid Argon Signal Processor*	A new component dealing with the digital processing of LAr calorimeter signals in the HL-LHC
LDPS	*LAr Digital Processing Board*	A Phase I upgrade that processes the digitized signal from the LTDB
LE	*Logic Elements*	A electronic device that consists of 4 Lookup tables and a D Flip-Flop. It performs simple logic operations
LHC	*Large Hadron Collider*	The most energetic proton collider in the world
LLC	*Logical Link Control*	Is a software implemented layer that acts as an interface between the MAC and the Network Layer
LPC	*Longitudinal Parity Checksum*	A checksum that is applied independently to each of a parallel group of bit streams
lpGBT	*Low Power Gigabit Transceivers*	CERNs custom communications protocol used in-between the FEB2-LASP interface

LS-x	*Long Shut Down x*	Period of operating breaks for the LHC when the machine is upgraded
LSB	*Layer Sum Boards*	Performs analogue summation of LAr calorimeter signals, by layer
LTDB	*LAr Trigger Digitizer Board*	A Phase I upgrade that continuously digitizes that Super cells signal
Luminosity $\mathcal{L}$		Instantaneous data rate or rate of collisions
LUT	*Lookup Table*	An table highlighting an output for different input combinations. This simple array indexing method saves computation runtime
M20K		Dedicated Memory RAM block of 20kB in Intel FPGAs
MAC	*Media/Medium Access Control*	A sublayer that controls the hardware responsible for interaction with the wired transmission medium
Macro blocks		see **Macro cells**
Macro Cells		A specialized group of cells that are fabricated at the transistor level on the FPGA for performing a specific task
Metastable state		A condition of a FPGA flip-flop where the output signal is in-deterministic
MGT	*Multi Gigabit Transceivers*	A data communications transceiver that can operate at rates above 1 Gbps
MLAB	*Memory Logic Array Block*	Superset of a LAB with specialized features for storing of data
mSGDMA	*Modular Scatter Gather DMA*	A DMA which is able to perform data movement operations with preloaded instructions, called descriptors
NIOS II		A soft-core processor which is programmed onto an FPGA
OSI	*Open Systems Interconnect*	A model that standardizes the communication functions of a computing system

PAM	*Pulse-Amplitude Modulation*	Signal modulation where the amplitude of a series of signal pulses encodes the message information
PCIe	*Peripheral Component Interconnect Express*	A bus interface standard used for connecting high-speed components
Phase I		First Phase of incremental upgrades towards HL-LHC
Phase II		Second and Final Phase of the incremental upgrades towards HL-LHC
PHY	*Physical Layer*	Connects the MAC to a physical medium such as copper cable
Pile up, $\langle\mu\rangle$		Average number of interactions per bunch crossing
PIO	*Parallel Input/Output*	An Intel core that provides an Avalon Interface to general-purpose I/O ports.
Platform Designer		An Intel Quartus Software tool which simplifies the design and creation of an Embedded system
PLL	*Phase Locked Loop*	A control system that generates an output with a phase related to the input. Generally used for creating clocks of different frequencies
QCD	*Quantum Chromodynamics*	A model that describes the strong interaction between quarks and gluons.
QSFP	*Quad Small Form-Factor Pluggable*	A networking interface with four lanes that can reach speeds of 25Gb/s
Quartus Prime		An Intel software suite used for programming and designing logic for Intel FPGA boards
RAM	*Random Access Memory*	Memory whose contents can be read, changed and reordered

RISC	*Reduced Instruction Set Computing*	A computer architecture that allows fewer cycles per instruction
ROD	*Readout Driver*	Performs the digital filtering and processing of the LAr calorimeter signals
SCA	*Switched-Capacitor Array*	A storage device that saves analogue readings from the calorimeter channel
SCFIFO	*Single Clock FIFO*	A First in First out buffer that has the same Input and Output clocks
SDRAM	*Synchronous Dynamic RAM*	A Random Access Memory device that stores each bit of data in a memory cell. Its operation is coordinated by a external clock.
SignalTap		A Quartus Prime Tool that can monitor the value of internal signals whilst the FPGA is running
SM	*Standard Model*	A theory used to describe the fundamental particles and fundamental forces
SoC	*System on a Chip*	An Integrated Circuit that includes a dedicated, prefabricated CPU
Socket (Ethernet)		Local endpoint of a network communication path
SRAM	*Static Random Access Memory*	A type of volatile memory where data is lost when power is switched off
Stratix 10		A family of Intel FPGAs which is used as a prototype for the LASP
Super Cells		An Phase I upgrade that results in finer granularity of the LAr calorimeter
TBB	*Tower Builder Boards*	All the pre-summed LAr calorimeter signals from the LSB are added up in the Tower Builder Boards
TCP	*Transmission Control Protocol*	A protocol describing how to establish and maintain network communications

TSE (IP)	*Triple Speed Ethernet IP*	An IP core provided by Intel which implements the Ethernet MAC and provides 10/100/1000Mbps Ethernet speeds
TTC	*Trigger, Timing and Control*	A system that performs clock management, triggering and monitoring status of subsystems
VL+	*Versatile Link+*	Optical transmission method between the FEB2-LASP interface

Preface

This thesis is a final work as Partial Fulfillment of the Requirements for the Degree of Masters of Science in the Department of Physics and Astronomy at the University of Victoria. Work in this thesis has been centered around designing and implementing an Injector Module for the ATLAS LASP (Liquid Argon Signal Processor) Firmware Group. A full working version of this project can be found in its dedicated GitLab Repository: `https://gitlab.cern.ch/atlas-lar-be-firmware/LASP/LASP-injector`

The Injector Project has been part of the contributions made by the University of Victoria to the LASP. It has been designed from scratch primarily by the author of this thesis, with the guidance of their supervisor, UVic Staff, and other LASP Firmware group members.

As is the case with embedded firmware designing, several Intellectual Property (IP) cores provided by third-party sources have been used in the Injector project design. All IPs mentioned in Chapter 6, except otherwise stated, are works obtained from Intel Corp. The interconnecting and interfacing of these IPs, however are works done by the author.

Setup of the NicheStack TCP/IP stack through software in Chapter 6.2.4 is works created by InterNiche Technologies, Inc and Intel Corp. The software principles of the socket server application was also derived from open source code created by Intel Corp. (then known as Altera Corporation). Specific applications done by the processor were modified to perform the functionalities needed by the Injector project. This was done by the author of the thesis.

The Injector project uses an instance of a Low power Gigabit Transceiver (lpGBT) emulator described in part of Chapter 6.5.2. This core has been created by the CERN lpGBT-FPGA group and the LASP Firmware group, specifically for projects like the Injector. The author of the thesis had no involvement in the creation of this module.

All the other work in the designing, implementing (Chapter 6) and testing (Chapter 7) of this project has been done by the author.

ACKNOWLEDGEMENTS

The work presented in this thesis would not be possible without the help of several people:

Professor R. Keeler, Thank you very much for your knowledge and guidance. Your farsightedness and project management has been a major factor for the completion of this work. Thank you very much for going beyond the call of duty to help support and mentor me. I very much appreciate it.

Professor R. McPherson, Thank you very much for your valued and expert input.

Dr. Sam de Jong, Thank you very much for helping me through the technical aspects of the project. I am in awe of your talent!

The ATLAS LASP Firmware group, Thank you very much for your guidance, suggestions and expertise. It has been an absolute pleasure to work in this group.

The UVic Physics community, From HEP-Experiment Faculty to the front-office staff, thank you so much for your constant encouragement and ideas. It has been very rewarding and fulfilling to be part of the community.

My office-mates, friends, and peers, Thank you for bearing me! The last few years has been nothing short of fantastic.

My parents and my family, For their never ending love.

May there be a thousand remedies,
ten thousand remedies!
Yasht 1.27

DEDICATION

To my sister,
Vahista.

Chapter 1

Introduction

The **Standard Model (SM)** of Particle Physics is the theory which so-far provides the best description of the fundamental constituents of matter and their interactions within a unified framework. Much of the current work done at the frontiers of Experimental Particle Physics concerns measuring and verifying different predictions given by the Standard Model. Deviations from expected values can open up doors to new physics beyond the Standard Model.

Particle colliders such as the **Large Hadron Collider (LHC)** are highly sophisticated ventures that are set up to collide energetic protons. General purpose experiments such as the **ATLAS (A Toroidal LHC ApparatuS)** detector are built around the LHC ring at collision points to thoroughly study and test various Standard Model Predictions. These multi-component machines are managed through the collaboration of scientists all around the world. Since no deviations from Standard Model predictions have been found yet, except the non-zero mass of the neutrinos, higher precision in the measurements needs to be obtained. This means that larger number of collisions or collisions at higher energy regimes are needed. Increasing the energy of collisions requires building a new accelerator or modifying parameters that were designed to be unchanged e.g. LHC ring diameter. These tasks would be time consuming and expensive.

Increasing the number of collisions, however, is an easier task with can be completed in a shorter time frame. Accordingly, an upgrade for the LHC, the **High-Luminosity LHC (HL-LHC)** is planned for 2025. The HL-LHC will result in an increase in proton-proton collisions by an order of magnitude.

Because of the higher luminosity, detector components need to be changed to incorporate a better trigger and a radiation-hard system. Current detector components

may not be able to select interesting events with enough efficiency and accuracy given the increased luminosity and background, which is why a better trigger system will be required. Further, a higher luminosity results in an increased particle flux which in turn increases the radiation damage. Current detector components will exceed their radiation limit during the operation of the HL-LHC. For these reasons, many detector components are also being upgraded.

One of the components of the ATLAS detector being upgraded is the **Liquid Argon (LAr)** calorimeter. The calorimeter is being renewed with advanced trigger and readout electronics. New **Front-End Boards (FEB2s)** are part of the new electronics that receive detector signals and digitize them continuously at 40 MHz. These signals are transmitted to another new component called the **Liquid Argon Signal Processor (LASP)** - the core of the back-end electronics. The LASP is responsible for receiving signals from the FEB2, applying digital filters and processing energy/time calculations, buffering data, and transmitting data to trigger systems. The LASP is implemented using **Field Programmable Gate Arrays (FPGAs)**. FPGAs are fitted into an FPGA board that allows the FPGA chip to use dedicated resources to perform communication and calculation tasks.

The functions of the LASP are broken down into different modules. Validation of individual modules of the LASP is of key importance in the development cycle. Additionally, verification of module behaviour with real ATLAS pulses will not be available until much later in the project timeline. Thus, there is a need for a test-bench that would inject ATLAS LAr FEB2 simulated pulses to the LASP.

The work of this thesis focuses on satisfying this need. A project called the "Injector" is implemented on an Intel Stratix 10 FPGA, housing a soft-core **NIOS II** processor that establishes a 1Gb/s **TCP/IP** communication protocol with a workstation. **ATLAS Readout Electronics Upgrade Simulation (AREUS)** Monte Carlo simulation pulses are generated on the workstation and transferred to the FPGA, where it is then stored in a 2GB DDR3 external memory chip. The pulses are then retrieved into internal memory buffers and are transmitted to the LASP at 40 MHz. The user is in complete control of the properties of the injected pulses. This vital property can test LASP behaviour for different cases and possible failure modes.

The description, implementation, and results of this project are expanded upon in this thesis. Chapter 2 provides a description of the Standard Model theoretical framework as well as some example physics goals that motivate the HL-LHC. The current LHC facility, as well as the ATLAS detector, are discussed in Chapter 3. Chapter

4 presents two phases that LHC will undergo in order to deliver the HL-LHC. This chapter also reports the changes in the ATLAS LAr system that will be made as the facility transitions into the High Luminosity environment.
In Chapter 5, an overall background to the Injector project, and its design specifications are given. An introduction to FPGAs as well as AREUS is also provided. An in-depth explanation of the design and implementation of the Injector project is given in Chapter 6. Chapter 7 presents the results of the developed project and also evaluates the overall project with respect to its design criteria.
A final summary and outlook towards the next generation of the Injector project are discussed in Chapter 8.

Chapter 2

The Standard Model of Particle Physics

The **Standard Model (SM)** of Particle Physics [1–5] is a relativistic quantum field theory that classifies all elementary particles and describes fundamental forces through which particles can interact. The SM theory was developed during the 1960s and 1970s, and has been experimentally verified to very high precision. A key success of the theory was the discovery of a particle called the Higgs Boson in 2012 [6].

The Standard model unifies the electromagnetic theory, the electroweak theory and the theory of strong interactions, leaving out the theory of gravity. Because of this, it is still considered to be an incomplete theory.

This chapter introduces the basic formulation of the model in Section 2.1. Phenomenology of the Standard Model Higgs is further discussed in Section 2.2

2.1 Foundational Theory

The Standard Model is a local gauge field theory based on the unitary product group

$$SU(3)_C \otimes SU(2)_L \otimes U(1)_Y \tag{2.1}$$

where the $SU(3)_C$ colour group governs the strong interaction acting on all particles with colour quantum numbers. Colour is the conserved quantity of this group. The $SU(2)_L \otimes U(1)_Y$ group governs the unified electroweak interactions. The conserved quantity of $SU(2)_L$ is called Weak Isospin (T_3) and the subscript L indicates that the symmetry only involves left-handed helicity particles. The subscript Y stands for

Weak Hypercharge.

2.1.1 Particle Constituents

According to the SM, matter is composed of 12 spin-$\frac{1}{2}$ *fermions*, 4 spin-1 *vector bosons* and 1 spin-0 *scalar Higgs boson* as shown in Figure 2.1. Fermions are further categorized as either *quarks* or *leptons* depending on the types of interactions they are subject to.

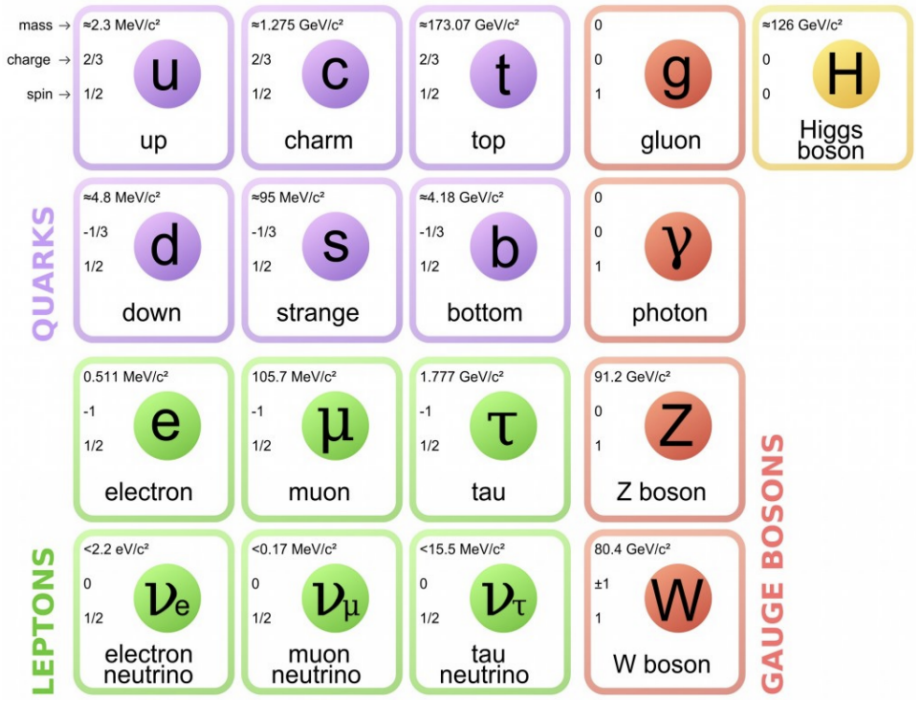


Figure 2.1: Particle content of the Standard Model (SM) with some of their associated quantum numbers and parameters. Each type of quark exists in three color charges. For each fermion, there is a corresponding anti-particle with exactly the same mass but with opposite quantum numbers.

Fermions appear in three generations with increasing mass. Each generation consists of a lepton pair and a quark pair. The lepton pair consists of a charged lepton and its associated electrically neutral *neutrino*. Charged leptons are subject to the electromagnetic and weak interactions, while the neutrinos are subject to only the weak interactions. The quark pair consists of an up-type quark and a down-type quark. The up-type quarks are up(u), charm(c) and top(t). The down-type quarks

are down(d), strange(s) and bottom(b). A list of all the fermion quantum numbers is shown in Table 2.1. Quarks are subject to the strong, weak and electromagnetic interactions.

		Electric charge Q	Weak isospin T_3	Weak hypercharge Y
Left-handed helicity fermions	ν_e, ν_μ, ν_τ	0	$+\frac{1}{2}$	-1
	e^-, μ^-, τ^-	-1	$-\frac{1}{2}$	-1
	u_L, c_L, t_L	$+\frac{2}{3}$	$+\frac{1}{2}$	$+\frac{1}{3}$
	d_L, s_L, b_L	$-\frac{1}{3}$	$-\frac{1}{2}$	$+\frac{1}{3}$
Right-handed helicity fermions	e_R^-, μ_R^-, τ_R^-	-1	0	-2
	u_R, c_R, t_R	$+\frac{2}{3}$	0	$+\frac{4}{3}$
	d_R, s_R, b_R	$-\frac{1}{3}$	0	$-\frac{2}{3}$

Table 2.1: Quantum numbers for fermions. No right handed neutrinos have been observed. The helicity is right-handed if the sign of the of the projection of the spin vector onto the momentum vector is positive, and vice-versa for left-handed helicity. Adapted from [4].

Fermions interact through the exchange of force-carrying particles (mediators), referred to as "gauge bosons". Mediators are interaction-specific. Interactions can only proceed if the fermions have the associated charge.
Gluons (g) are spin-1, massless mediators of the strong interaction. Colour is needed for strong interactions to take place and so quarks (and gluons themselves) are the only elementary particles able to be mediated by gluons. The mediator of the electromagnetic interaction is the neutral spin-1 massless photon (γ). Electrical charge (Q) is needed and so only neutrinos do not interact electromagnetically and thus do not interact with the photon. The weak interaction is mediated by either the massive $W^\pm$ boson or the massive neutral Z boson. Since all fermions (except for right-handed neutrinos[1]) contain weak-hypercharge, they are able to undergo weak interactions.

The spin-0, electrically neutral scalar Higgs boson is responsible for generating the masses of the massive gauge bosons and the elementary particles within the theory. Fermions couple to the scalar boson proportionally with its mass and couple with a vector boson proportionally to the square of its mass.

[1] if existent

2.1.2 Quantum Chromodynamics

Quantum Chromodynamics (QCD) [7,8] described within the $SU(3)_C$ symmetry group, governs how the strong interactions between quarks are mediated by gluons. There are three associated charges for this group known as the colour charge. Gluons can be thought of as having both colour and anti-colour charges. Since the strong interaction with infinite range is not observed, the colour singlet combination of the gluon does not exist. There are thus a total of $3^2 - 1 = 8$ gluon configurations.

Since the gluons themselves have colour charge, they are able to interact not only with quarks but also among themselves, thus leading to three or four-gluon vertices. Possible interactions within QCD are shown in Figure 2.2 as Feynman diagrams.

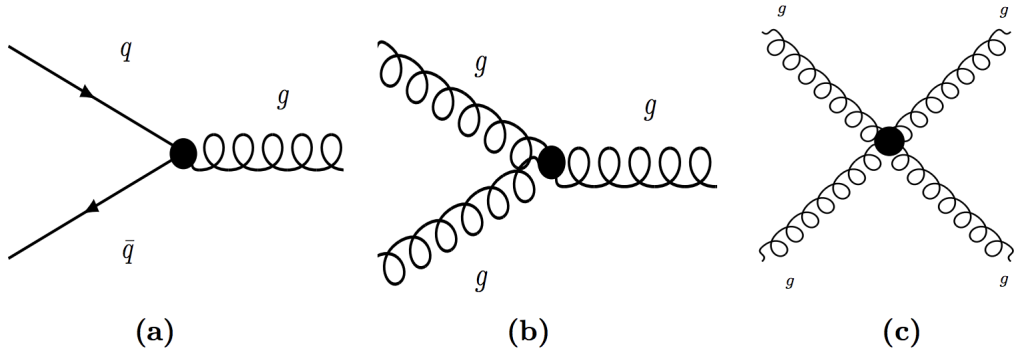


Figure 2.2: Possible interaction vertices within QCD: (a) shows a quark-gluon interaction; (b) shows a three gluon vertex and (c) shows a four gluon vertex. Adapted from [9].

The self interaction of the gluons provide a unique effect of "anti-screening" that is manifested in QCD. For short distances, the coupling strength between quarks decreases and the quarks can be thought to behave as free particles. The particles are said to be *asymptotically free*. On the other hand, the coupling strength increases for large distances, thus making it impossible for quarks to be isolated. Quarks are therefore always bound into hadrons, in a property known as *confinement*.

2.1.3 Electroweak theory

The electroweak theory [8, 10, 11] described by the Glashow-Salam-Weinberg (GSW) theory, unifies the electromagnetic and weak interactions. This is done using the gauge symmetry group $SU(2)_L \otimes U(1)_Y$. The unification causes the electric charge, Q, to be related to the weak hypercharge Y (i.e. the associated charge of $U(1)_Y$)

and weak isospin T_3 (i.e. the associated charge of $SU(2)_L$) via the non-trival linear combination:

$$Q = T_3 + \frac{1}{2}Y \tag{2.2}$$

The gauge fields of the $SU(2)_L \otimes U(1)_Y$ symmetry group correspond to four massless mediating bosons, organised in a weak isospin triplet W^1, W^2, W^3 ($SU(2)_L$) and a weak hypercharge singlet B ($U(1)_Y$).

Since the electroweak theory has to be invariant under global and local $SU(2)_L \otimes U(1)_Y$ transformations, the covariant derivative of the theory is written as:

$$D_\mu = \partial_\mu + ig'\frac{Y}{2}B_\mu + ig\frac{\sigma_a}{2}W_\mu^a \tag{2.3}$$

where g' and g are the coupling constants for $U(1)_Y$ and $SU(2)_L$, W_μ^a and B_μ are the gauge bosons of the $SU(2)_L$ and $U(1)_Y$ groups, respectively. The σ_a ($a = 1, 2, 3$) represent the Pauli matrices.

Parameters of the unified theory can be related to the coupling constants of $SU(2)_L$ and $U(1)_Y$. The electric charge Q, for example is given by

$$Q = g \sin\theta_W = g' \cos\theta_W \tag{2.4}$$

where θ_W is the weak mixing angle, also called the Weinberg angle.

The mixing also causes what will become the physical gauge bosons: γ, Z, $W^\pm$ to arise from a combination of the W and B fields. This can be expressed as:

$$A_\mu = W_\mu^3 \sin\theta_W + B_\mu \cos\theta_W \tag{2.5}$$

$$Z_\mu = W_\mu^3 \cos\theta_W - B_\mu \sin\theta_W \tag{2.6}$$

$$W_\mu^\pm = \frac{1}{2}\left(W_\mu^1 \mp W_\mu^2\right) \tag{2.7}$$

where the mathematical expression A_μ can be identified with the physical gauge boson of the photon γ.

2.1.4 Spontaneous Symmetry Breaking and the Higgs Mechanism

The electroweak theory and QCD only allow massless particles. The Brout-Englert-Higgs model [12–14] was formulated in 1964 as a way to solve the mass problem. The model introduces a new scalar boson, the *Higgs Boson*, with the mechanism of spontaneous electroweak symmetry breaking.

The Lagrangian of the Higgs field, Φ, can be written as

$$\mathcal{L}_{Higgs} = (D_\mu \Phi)^\dagger D^\mu \Phi - V(\Phi) \tag{2.8}$$

where D_μ is the covariant derivative of $SU(2)_L \otimes U(1)_Y$ and $V(\Phi)$ is the Higgs potential.

Two free parameters, μ, λ are introduced to define the Higgs potential:

$$V(\Phi) = \mu^2 \Phi\Phi^\dagger + \lambda \left(\Phi\Phi^\dagger\right)^2 = \mu^2 \Phi^2 + \lambda \Phi^4 \tag{2.9}$$

To have a stable theory, the potential must be have a stable minimum. This imposes $\lambda > 0$. The sign of μ^2 determines how the shape of the potential looks like, as shown in Figure 2.3:

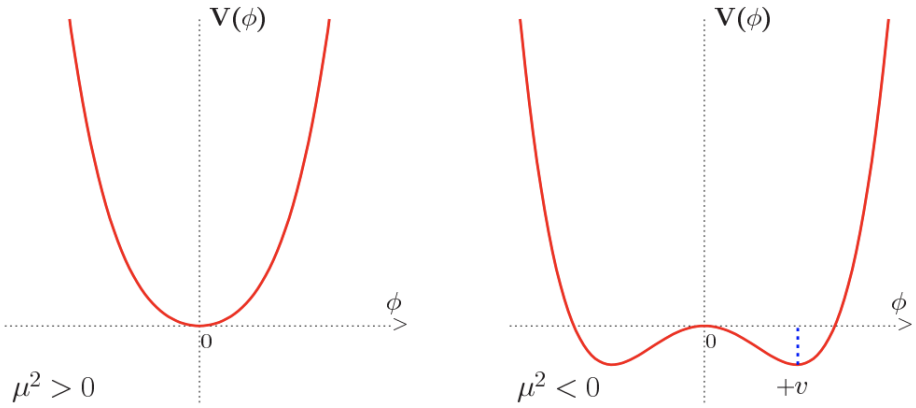


Figure 2.3: The shape of the potential $V(\Phi)$ in the case of $\mu^2 > 0$ (left) and $\mu^2 < 0$ (right). The vacuum expectation value, $v \approx 246$ GeV. Diagram adapted from [9].

- if $\mu^2 \geq 0$ the potential has a parabolic shape with a trivial minimum at $\Phi = 0$

- if $\mu^2 < 0$ the potential has a local maximum at $\Phi = 0$ and an infinite number of degenerate states of minimum energy satisfying:

$$|\Phi_0| = \sqrt{\frac{-\mu^2}{\lambda}} = v \approx 246 \text{ Gev} \tag{2.10}$$

  where Φ_0 is the value of the field in the minimum of the potential, also known as the vacuum expectation value (vev).

In the case of $\mu^2 < 0$, the ground states are not invariant i.e. transformations applied to one ground state will rotate it to a different physical state. The symmetry is thus spontaneously broken.

To further illustrate this, an analogy of magnetic domains can be used. A piece of iron with a temperature above the Curie point will not have a ferromagnetic moment. Application of a magnetic field in a particular direction will result in a paramagnetic field in that direction. If the iron is now allowed to cool below the Curie point in zero external magnetic field and is further tapped, ferromagnetic domains will form and start to align in a single direction. Since there are multiple directions in which the ferromagnetic domains could be lined up, each with exactly the same ground state energy, the act of the iron's magnetic domain lining up to one direction is random and happens because the symmetry is now broken. While the ground state energy is the same regardless of the direction of the magnetization, the energy of the system will depend on the direction when an external field is applied. The direction of the magnetization represents a particular physical state.

The strength of the coupling of the Higgs field with a particle fixes the mass of the particle. Boson masses can be deduced from the theory to be

$$m_H = \sqrt{2\lambda}v \quad m_W = \frac{gv}{2} \quad m_Z = \frac{gv}{2\cos\theta_W} \quad m_\gamma = 0 \tag{2.11}$$

The masses of the fermions can be added to the theory by introducing a mass parameter for each fermion

$$m_f = \frac{1}{\sqrt{2}} v y_f \tag{2.12}$$

where y_f is the fermion specific Yukawa couplings.

In addition, the Lagrangian for the Higgs field contains triple and quartic coupling terms of the field with the $V = W^{\pm}, Z$ bosons (λ_{HVV} and λ_{HHVV}, respectively) , and also contains triple and quartic Higgs boson self-coupling terms (λ_{HHH} and λ_{HHHH},

respectively). The strength of these terms are:

$$\lambda_{HVV} = \frac{2m_V^2}{v} \quad \lambda_{HHVV} = \frac{m_V^2}{v^2} \quad \lambda_{HHH} = \frac{m_H^2}{2v} \quad \lambda_{HHHH} = \frac{m_H^2}{8v^2} \tag{2.13}$$

Figure 2.4 shows the Higgs couplings permitted.

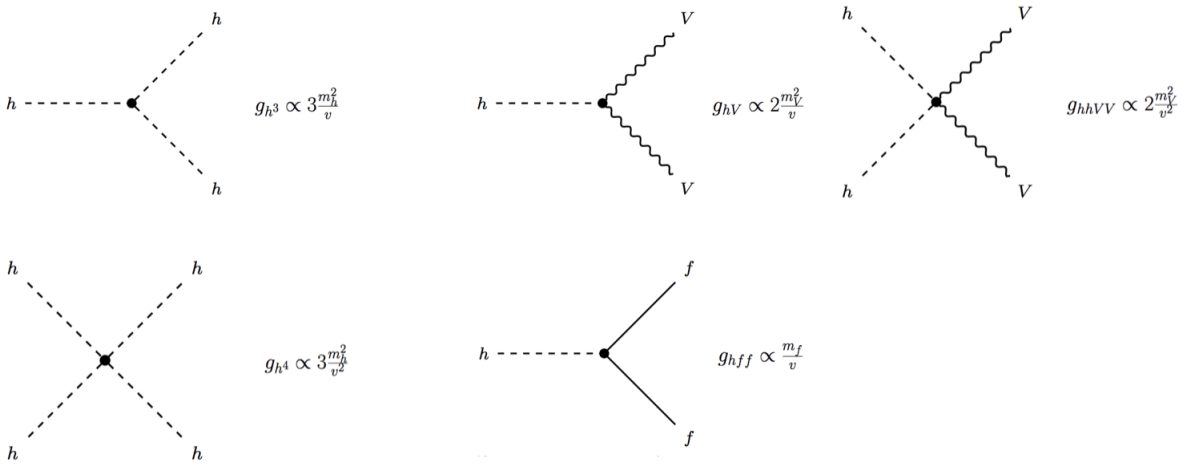


Figure 2.4: Couplings of the Higgs boson in the Standard Model. Diagram adapted from [15].

2.2 Phenomenology of the Standard Model Higgs Boson

2.2.1 Production in proton-proton collisions

Since the work of this thesis focuses on the LHC collider, proton-proton collisions are examined as a way to produce the SM Higgs boson. As mentioned, the coupling of the Higgs Boson to particles is proportional to the particle mass, and to vector bosons is proportional to mass squared. As expected, the production of the Higgs boson is dominated by processes involving heavy particles such as the 3rd generation quarks and the massive vector bosons. The main production mechanisms are gluon fusion (ggF), vector boson fusion (VBF), production in association with a W or Z boson (WH or ZH) and the associated production with top (ttH) or bottom (bbH) quark pairs or with single top quarks (tH) [8]. Leading order Feynman diagrams of

the processes are shown in Figure 2.5. The cross sections of the dominant production modes of the Higgs as a function of center of mass energy, $\sqrt{s}$, are shown in Figure 2.6. The dominant production mode with a relative rate of 88% is the ggF process.

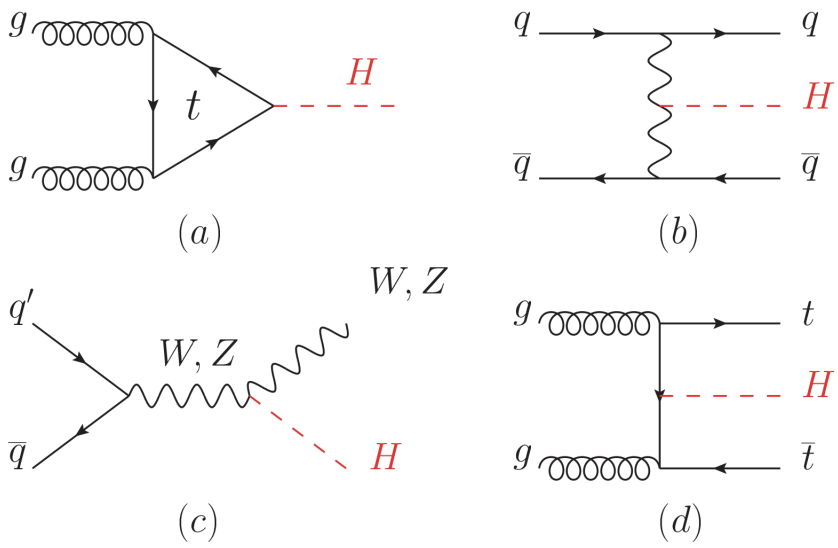


Figure 2.5: Main leading order Feynman diagrams contributing to the Higgs production in (a) gluon fusion, (b) Vector-boson fusion, (c) associated production with a gauge boson, (d) associated production with a pair of top (or bottom) quarks. Diagram taken from [8].

Because the gluons cannot directly couple to the Higgs boson, an intermediate quark loop is needed.

Another important prediction of the theory is the production of a Higgs boson pair ("Double-Higgs Production"). This process is of interest because it enables further probing of the Brout-Englert-Higgs Model. The leading order Feynman diagrams contributing to the gluon fusion are shown in Figure 2.7. The quark loop could either be a triangle or a box. The loops are dominated by the top quark since it is the heaviest quark (and so has the largest coupling to the Higgs boson).

The triangular diagram is due to an interesting tri-linear coupling of the Higgs, whose coupling strength λ_{HHH} provides valuable information about the shape of the Higgs potential and is a critical test as to whether the Higgs boson discovered in 2012 at CERN is the one predicted by the Brout-Englert-Higgs mechanism.

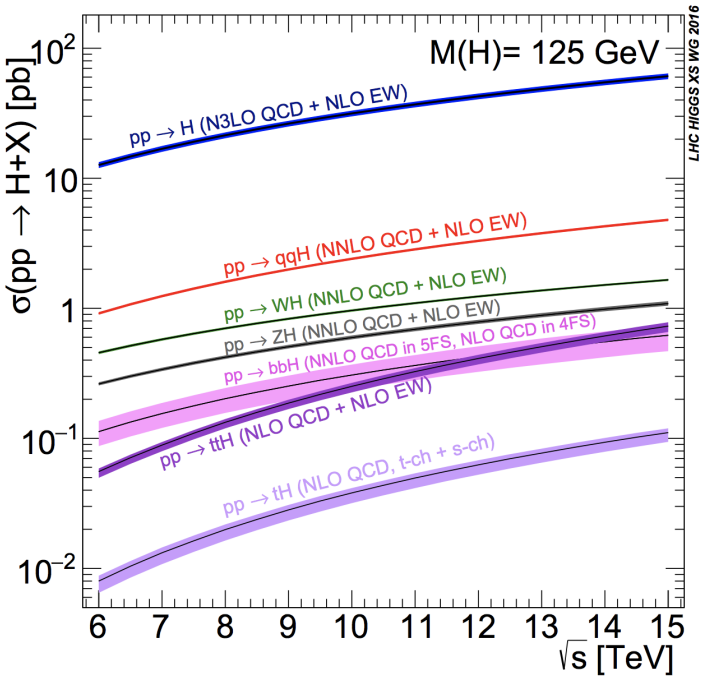


Figure 2.6: Cross sections of the dominant production modes of the Standard Model Higgs boson with a mass of $m_H = 125$ GeV as a function of the LHC centre-of-mass energy. Diagram taken from [16].

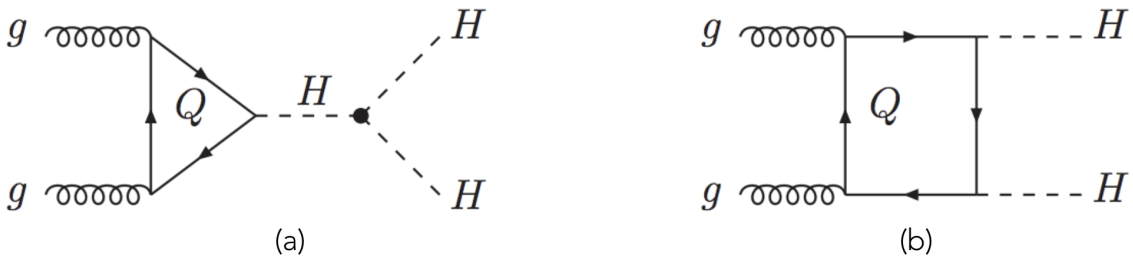


Figure 2.7: Leading order Feynman diagrams involved in the Higgs boson pair via gluon fusion. The production occurs by either a (a) triangular quark loop or a (b) box quark loop. Diagram taken from [17]

2.2.2 Decay modes and the hunt for λ_{HHH}

In order to measure the di-Higgs production in a detector, their decay products are observed. The Higgs boson has various decay modes as shown in Table 2.2. The choice of which channel to observe is usually a compromise between event rate and signal extraction quality. The abundant $H \to b\bar{b}$ suffers from having a large multijet background that makes it harder to extract the signal. The cleanest signal is from

the $H \rightarrow \gamma\gamma$ decay which provides a very narrow mass peak. However, a very low expected yield prevents this from being a viable solution. The Higgs boson pair

Decay Channel	Branching Ratio (%)	Total Yield at 139 fb^{-1}	Total Yield at 3000 fb^{-1}
$b\bar{b} + b\bar{b}$	33	1 900	40 000
$b\bar{b} + W^+W^-$	25	1 400	31 000
$b\bar{b} + \tau\bar{\tau}$	7.3	400	8 900
$b\bar{b} + ZZ$	3.1	170	3 800
$\tau\bar{\tau} + W^+W^-$	2.7	150	3 300
$ZZ + W^+W^-$	1.1	60	1 300
$b\bar{b} + \gamma\gamma$	0.26	10	320
$\gamma\gamma + \gamma\gamma$	0.0010	0	1.2

Table 2.2: Table of common decay channels for a Higgs boson pair, with the corresponding branching ratio and approximate yield at 139 fb^{-1} and 3000 fb^{-1} before any event selection is applied. A total production cross-section of 40.8 fb is assumed. Adapted and Modified from [18].

production is a very rare process which has not yet been observed in the most recent data of the LHC. Table 2.2 also shows the total expected number of events for two situations. For an integrated luminosity (further explained in Chapter 3) of 139 fb^{-1}, it can be noted that the event count is very low for any significant measurement to take place. This is the current situation of the LHC. With an increased integrated luminosity, a much larger event yield can be obtained. The integrated luminosity of 3000 fb^{-1} is planned for an upgraded version of the LHC, the so called "HL-LHC".

Measuring the cross-section for the di-Higgs production and comparing it with its standard model prediction gives insight to the shape of the Higgs potential, the validity of the current standard model and perhaps, if applicable, new information to Beyond the Standard Model (BSM) processes. This is a prime physics motivation to upgrade the current parameters of the LHC.

Chapter 3

The Large Hadron Collider and ATLAS

The **Large Hadron Collider (LHC)** is a circular proton-proton collider located at the **Conseil Européen pour la Recherche Nucléaire (CERN)** in Geneva, Switzerland. Spanning a circumference of about 27 km, it is currently the largest and most powerful particle accelerator ever since it began operations in September 2008. Four major and three minor experiments are housed within the LHC ring namely: ALICE, **ATLAS**, CMS, LHCb, LHCf, MoEDAL and TOTEM. A description of the accelerator is given in Section 3.1. The focus of this thesis is directed towards the upgrade of the ATLAS experiment. This experiment and its component detectors are described in Section 3.2.

3.1 CERN's Accelerator Complex and the LHC

Located in a tunnel between Lake Geneva and the Jura mountains, the 27 km long LHC accelerates protons to a nominal energy of 6.5 TeV per beam, yielding a center of mass energy of 13 TeV when the protons collide head-on.

Before particle bunches are accelerated in the LHC, they pass through a chain of pre-accelerators as shown in Figure 3.1. Proton beams are initially accelerated to 50 MeV by the Linac II accelerator, after which they pass through the booster attaining energies of 1.4 GeV. Next, the protons are injected into the Proton Synchrotron (PS) and then the Super Proton Synchrotron (SPS) through which they gain energies of 450 GeV. The protons are then injected into one of the LHC's rings.

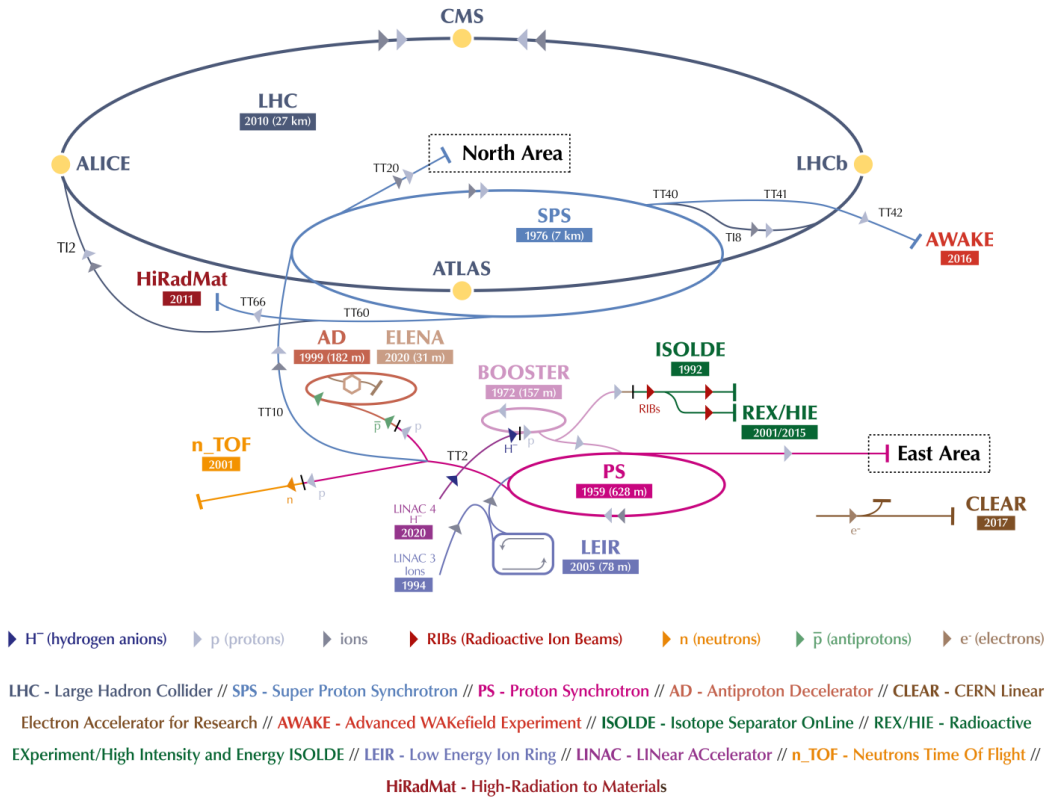


Figure 3.1: CERN accelerator complex illustrating the LHC, its preaccelerators and its major experiments. Diagram taken from [19].

There are two counter-rotating beams in the LHC. The particles in the beams are bunched by eight radio-frequency accelerating cavities per beam. In the nominal design, each bunch may contain up to 1.15×10^{11} protons, and the LHC contains 2808 bunches per beam that are separated by 25 ns (40 MHz bunch crossing rate) [20].

Along the circumference of the LHC are dipole magnets as well as quadrupoles which are used for beam bending and beam focusing/steering respectively. 1232 superconducting dipole magnets, each 14.3m long, are cooled to 1.9 K using liquid helium and can provide a magnetic field strength of 8.33 T [20]. The magnetic field in the dipoles is directed perpendicular to the beam so that it can bend it and keep it on track.

Particle bunches in the beam collide at four points in the LHC, each of which houses one of the major experiments. ATLAS and CMS are multi-purpose detectors capable of searching for a wide variety of processes eg. the Higgs boson, SUSY particles, etc. LHCb focuses on B-physics, particularly on CP-violation. ALICE aims to measure the properties of quark-gluon plasma in ion-ion collisions.

An accelerator can be characterized by two main quantities: the energy of accelerated particles and the luminosity. The **luminosity,** $\mathcal{L}$, of an accelerator determines the rate of particle interactions, R, occurring for a given process with cross section σ:

$$R = \mathcal{L}\sigma \tag{3.1}$$

The luminosity of the machine depends only on the beam parameters and is defined as the beam intensity I per collision area A. Assuming a Gaussian beam profile, the luminosity of a circular accelerator for two oppositely directed beams can be expressed as

$$\mathcal{L} = \frac{I}{A} = \frac{f_{rev} n_b N_1 N_2}{A} \tag{3.2}$$

where N_1 and N_2 are the number of particles in each bunch per beam, n_b is the number of bunches in either beam around the ring, A is the cross-sectional area of the beams, and f_{rev} is the revolution frequency [3].

This can be made more specific for the LHC case:

$$\mathcal{L} = \frac{f_{rev} n_b N_1 N_2}{4\pi\sigma_x\sigma_y} \cdot R_\phi \tag{3.3}$$

where $\sigma_{x,y}$ denote the horizontal and vertical convolved beam widths and R_ϕ is the geometrical loss factor ($R_\phi < 1$) that takes into account the non-zero crossing angle at the interaction point.

The LHC is designed to run at $\mathcal{L} = 10^{34}\text{cm}^{-2}\text{s}^{-1}$ [20]. The total number of interactions, called **integrated luminosity,** L, is obtained by:

$$L = \int \mathcal{L} dt \tag{3.4}$$

When particle bunches cross one another in the LHC, more than one p-p collision happens. A parameter called **pile-up,** $\langle\mu\rangle$, quantifies the average number of interactions per bunch crossing and is given by:

$$\langle\mu\rangle = \frac{\mathcal{L}\sigma_{tot}}{n_b f_{rev}} \tag{3.5}$$

where σ_{tot} is the total cross-section for a p-p interaction.

3.2 The ATLAS experiment

ATLAS is one of the four main experiments located at the **LHC**. It is a general-purpose physics experiment designed to test the complete range of physics opportunities that the LHC provides. Together with CMS, another general-purpose experiment based at the LHC, ATLAS achieved a significant milestone with the discovery of the Higgs boson in 2012 [6].This section describes the components of the ATLAS detector with a particular emphasis on the calorimeters.

The ATLAS detector [21,22], shown in Figure 3.2, spans 44 m in length and 25 m in width and height. It is placed in an underground cavern and the total weight of the detector is about 7000 t. The detector covers the full solid angle.

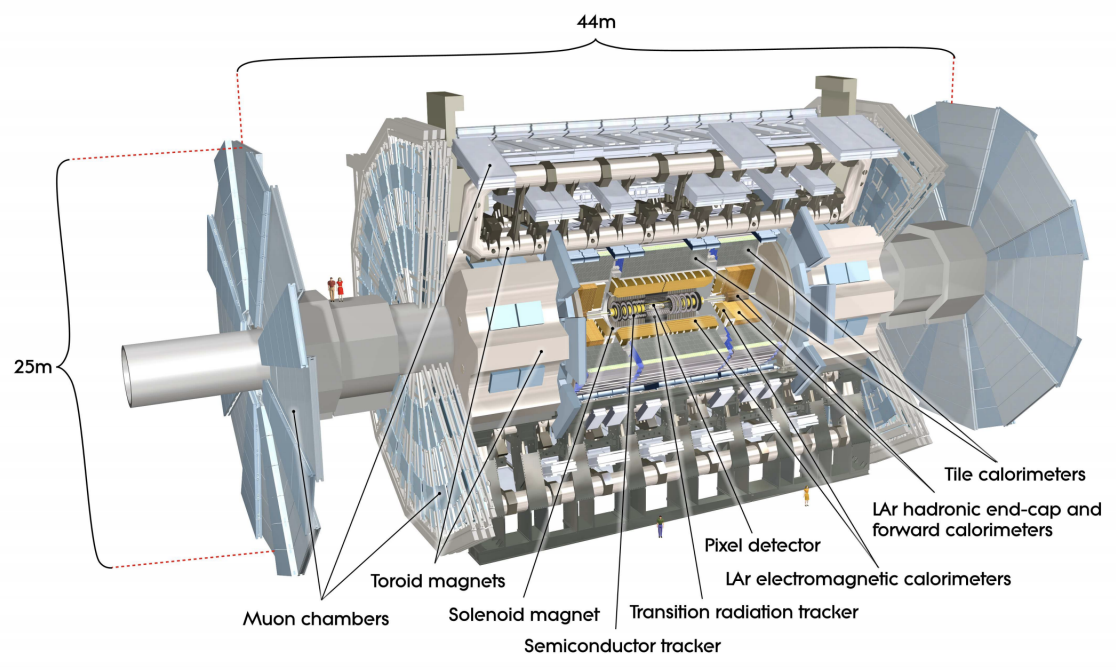


Figure 3.2: Schematic view of the ATLAS detector. Taken from [23].

ATLAS uses a spherical-polar coordinate system with the z-axis oriented along the beam pipe. $z = 0$ is located at the center of the detector. The azimuthal angle ϕ is measured around the beam pipe, with $\phi = 0$ pointing towards the center of the LHC ring. The polar angle θ is measured from the beam axis. A more convenient parameter called pseudorapidity η, defined as $\eta = -\ln \tan \theta/2$, is used instead of the polar angle θ because the difference in rapidity is Lorentz invariant.

The ATLAS detector is composed of sub-detectors, each focusing on measuring

and detecting different particles that may appear out of the p-p collisions. From the inside to the outside the sub-detectors are: Inner detector, Calorimeter, and Muon Spectrometer.

3.2.1 The Inner Detector

The Inner Detector (ID) has dimensions of about 6.2 m in length and 2.1 m in diameter [23], and focuses on the tracking of charged particles resulting from the collisions. The Inner Detector is further broken down into three different sub detectors namely, in order from inner to the outer layer, the pixel detector, the semiconductor tracker (SCT), and the transition radiation tracker (TRT).

The Pixel detector, located at the innermost layer, receives the highest particle flux. It is here where the first measurements of track momentum, vertex position as well as impact parameter are taken. The pixelated nature of the readout electrodes offers the highest granularity of the whole ATLAS detector.

The SCT uses strips instead of the pixel electrodes in the previous layer. The strips are made out of silicon electrodes. Since the strips only provide tracking information in one dimension, two layers are used with a slight tilt with respect to each other in order to provide two dimensionality tracking. The strips are widely used because this geometry reduces the number of readout channels required thus ending up being a much cheaper option than the pixel detector. The SCT is organized in four barrel layers and nine end cap layers on either side. Each of the barrel layers is 149.8 cm along the beam pipe and mounted at a radii of 29.9 cm, 37.1 cm, 44.3 cm and 51.4 cm around the beam pipe. The SCT covers an overall range up to $\mid \eta \mid < 2.5$.

The TRT is comprised of xeon (and argon) gas filled tubes that are interleaved within polypropylene fibres. When a relativistic particle crosses an interface of two materials with different dielectric constants, transition radiation occurs. This offers the TRT ability for pattern recognition and significantly contributes to the particle identification. In total, the TRT consists of approximately 351,000 tubes organized in 73 layers in the barrel and 160 layers in the end cap region. The tubes have a diameter of 4mm and a length of 144cm and 37cm for the barrel and end-cap region, respectively. The TRT is able to provide a coverage for $\mid \eta \mid < 2$.

3.2.2 Calorimeters

The ATLAS calorimeters are located next to the Inner Detector. Their main purpose is to measure the energy of incoming particles. To do this, they stop and absorb the particles, forcing them to deposit their energy within the detector. Two Standard Model particles normally pass through the calorimeters without stopping: muons and neutrinos. For the detection of muons, a special muon spectrometer is built. The detection of neutrinos and other non-interacting particles is inferred from the missing transverse energy E_T^{miss}, which is an imbalance in the vector sum of the momenta in the plane transverse to the beam axis. The calorimeter system in ATLAS is broken down into two parts: an electromagnetic and a hadronic calorimeter. The electromagnetic calorimeter measures the energy of electrons and photons (and part of the energy of the protons and hadrons). The hadronic calorimeter measures the energy of protons and hadrons. The schematic view of the calorimeter system is shown in Figure 3.3.

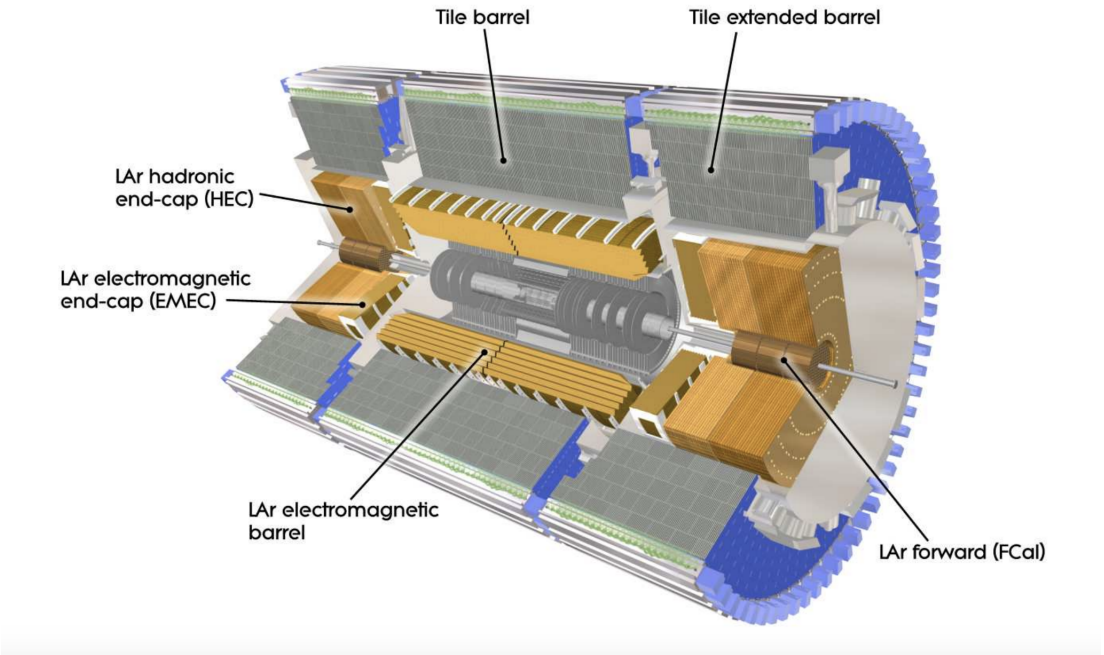


Figure 3.3: Schematic view of the ATLAS calorimeters. The grey area in the center shows the Inner Detector. Diagram taken from [19].

The Electromagnetic Calorimeter

The **Electromagnetic Calorimeter (ECAL)** is placed next to the ID's TRT with a solenoidal magnet in-between. The electromagnetic calorimeter is composed of the **Electromagnetic Barrel (EMB)** and the **Electromagnetic end-caps (EMEC)**. The ECAL is a sampling calorimeter which means it consists of an absorbing and a detecting (sampling) material. Lead is used as an absorber and **Liquid Argon (LAr)** is used as the sampling material.

The LAr gets ionized as a charged electron passes through it. The electrons and ions drift to electrodes, which have a voltage difference of about 2kV, causing a signal. The lead absorber material in the ECAL has accordion geometry which allows for segmentation in the radial direction. The accordion shape folding angle changes with distance from the interaction point, with layers closer to the interaction point having higher segmentation to increase resolution. The EMEC is also divided into two: an inner wheel endcap and an outer wheel endcap. The difference between them is that the inner wheel has a coarser granularity and two active layers instead of three. The full ECAL covers a range of up to $\mid \eta \mid < 3.2$.

Because the LAr calorimeter is a core subject matter of this thesis work, a more detailed treatment is give in Section 3.3

The Hadronic Calorimeter

The hadronic calorimeter is composed of three sub-detectors: the Tile calorimeter, the Hadronic End-cap Calorimeter (HEC), and the forward calorimeter (FCal). The tile calorimeter is also a sampling calorimeter using steel plates as the absorber and scintillator as the active part. Scintillating light produced is shifted by wavelength shifting fibres and is directed to photomultipliers. The tile calorimeter extends up to $\mid \eta \mid < 1.7$.

The HEC is located directly behind the EMEC and covers a range of $1.5 < \mid \eta \mid < 3.2$. It is composed of two wheels on each side, segmented in depth into two parts each. The absorbing material is copper plates placed with LAr acting as the active medium. The ionization signal is read out via electrodes placed in the middle of the 8.5mm gap between the two copper plates.

The FCal is the most forward part of the calorimeter covering a range of $3.1 < |\eta| < 4.9$. This part of the calorimeter is very close to the beam axis and experiences high particle flux. To accommodate this, the design of the FCal consists of copper

and tungsten layers in depth that serve as the absorbing material. Cylindrical holes oriented parallel to the beam axis are arranged in a matrix. Copper and tungsten rods are centred in the holes leaving a very thin cylindrical gap filled with LAr. The signal is read out at the ends of the rods which act like the centre of a coaxial cable.

3.2.3 The Muon Spectrometer

The Muon spectrometer is the outermost detector layer in the ATLAS system and is also the largest in terms of physical size. Since many of the detectable particles are already absorbed in the calorimeter, the muon spectrometer only detects the remaining muons. Tracking of the trajectory of muons as accurately as possible is important for yielding a fast trigger signal. A schematic view of the muon spectrometer is shown in Figure 3.4. The muon spectrometer consists of a magnet system and

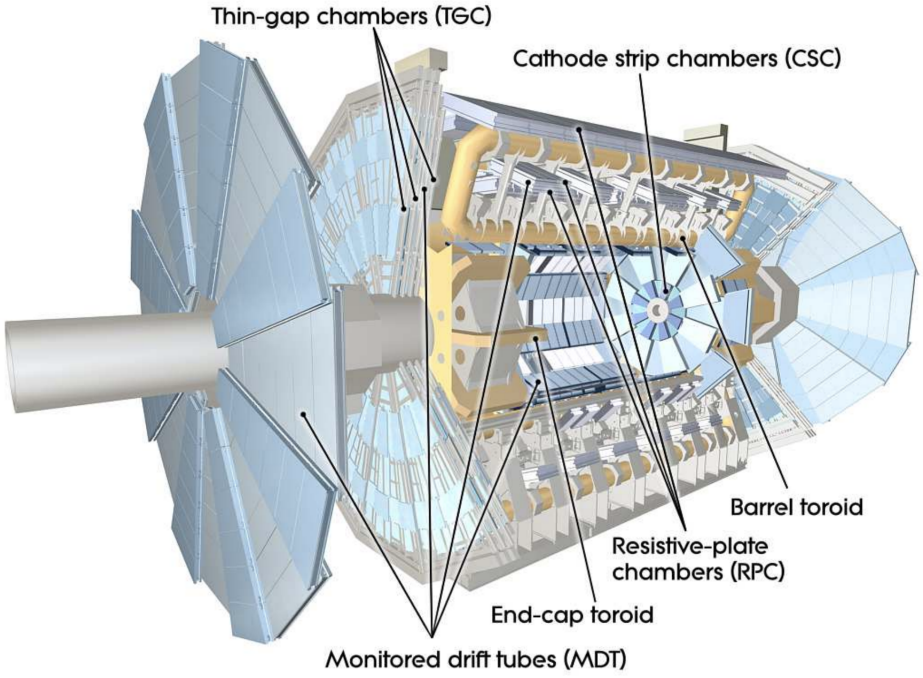


Figure 3.4: Schematic view of the ATLAS muon spectrometer. Diagram taken from [23].

four sub-detectors: the Monitored Drift Tubes (MDT), the Cathode Strip Chambers (CSC), the Resistive Plate Chambers (RPC), and the Thin Gap Chambers (TGC). The MDTs track muons up to $|\eta| < 2.7$ and offer precision measurement of momentum and tracking of muons. The system of CSCs covers the highest range of

pseudo-rapidity, $2 < |\eta| < 2.7$, and provides precision tracking information in the forward direction. The RPCs are used in the barrel region until $|\eta| < 1.05$ and provide tracking information. The TGCs are located in the end cap region offering granularity in the azimuthal part. Their coverage extends the range to $1.05 < |\eta| < 2.7$

3.2.4 Trigger System

If an event size is approximated to be 1.3 Mbytes, the LHC running at 40 MHz would cause the ATLAS detector alone to produce about 50 Tbytes per second. This massive quantity of information is too large to store or even transmit. For this reason, only selected events are stored whilst the other non-interesting events are discarded. A trigger system decides which events will be kept and which ones will be discarded.

The ATLAS trigger system is composed of two different components: a hardware based first Level trigger (Level 1) and a software-based **High Level Trigger (HLT)**. Each level is responsible for further reducing the event rate. At each level of progression, more information from different parts of the detector is used to make a trigger decision.

The Level 1 trigger only uses information from the muon system and parts of the calorimeter, but no tracking information from the Inner Detector. It reduces the rate from 40 MHz to ≈100 kHz. Hit information from the Inner Detector is used for the HLT, only if the event has passed the Level 1 trigger. The event filter step is done offline and the event rate is reduced to about 1 kHz.

3.3 LAr Calorimeters

The **LAr calorimeter** uses Liquid Argon as the active medium and is found in the EMB in the center, the front and back EMECs, the FCal, and the HEC.

A total of 182468 detector cells compose the LAr calorimeters [23]. As an ionizing particle passes through the LAr, electrons drifting to the electrodes produce an analogue electrical signal, proportional to the energy deposition of the particle. The cells are arranged in layers, each with its distinct granularity and geometry as shown in Figure 3.5

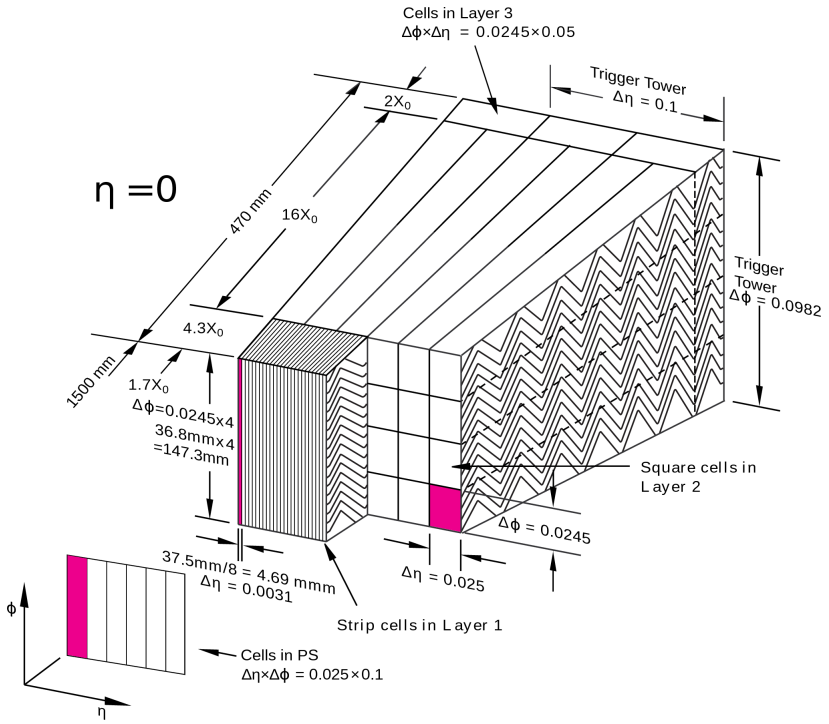


Figure 3.5: Schematic view of the LAr calorimeter barrel showing the arrangement of single cells in different layers. Diagram taken from [24].

3.3.1 Existing Readout systems

Electrical signals produced from the cells are read out, digitized, and processed. The readout system is broken into a radiation-hard front-end system mounted directly on the cryostat, and a back-end system which is located off of the detector.

One of the main electronics of the front-end system is the readout **front-end-board (FEBs)** which are designed to read out and digitize the LAr calorimeter signals without degrading the energy resolution. Up to 128 channels are grouped and processed by each of the 524 FEBs.

One of the main tasks of the back end system is to perform digital filtering of signals based on **Digital Signal Processors (DSPs)**. Calibrated quantities from the back-end are then sent via optical fibres to the **data acquisition (DAQ)** system. Figure 3.6 gives an overview of the current LAr Electronics data path.

3.3.2 Signal Processing by LAr Electronics

FEBs perform the first level of signal processing including pre-amplification, shaping, analogue buffering, digitisation, and gain-selection of the detector input signal. FEBs

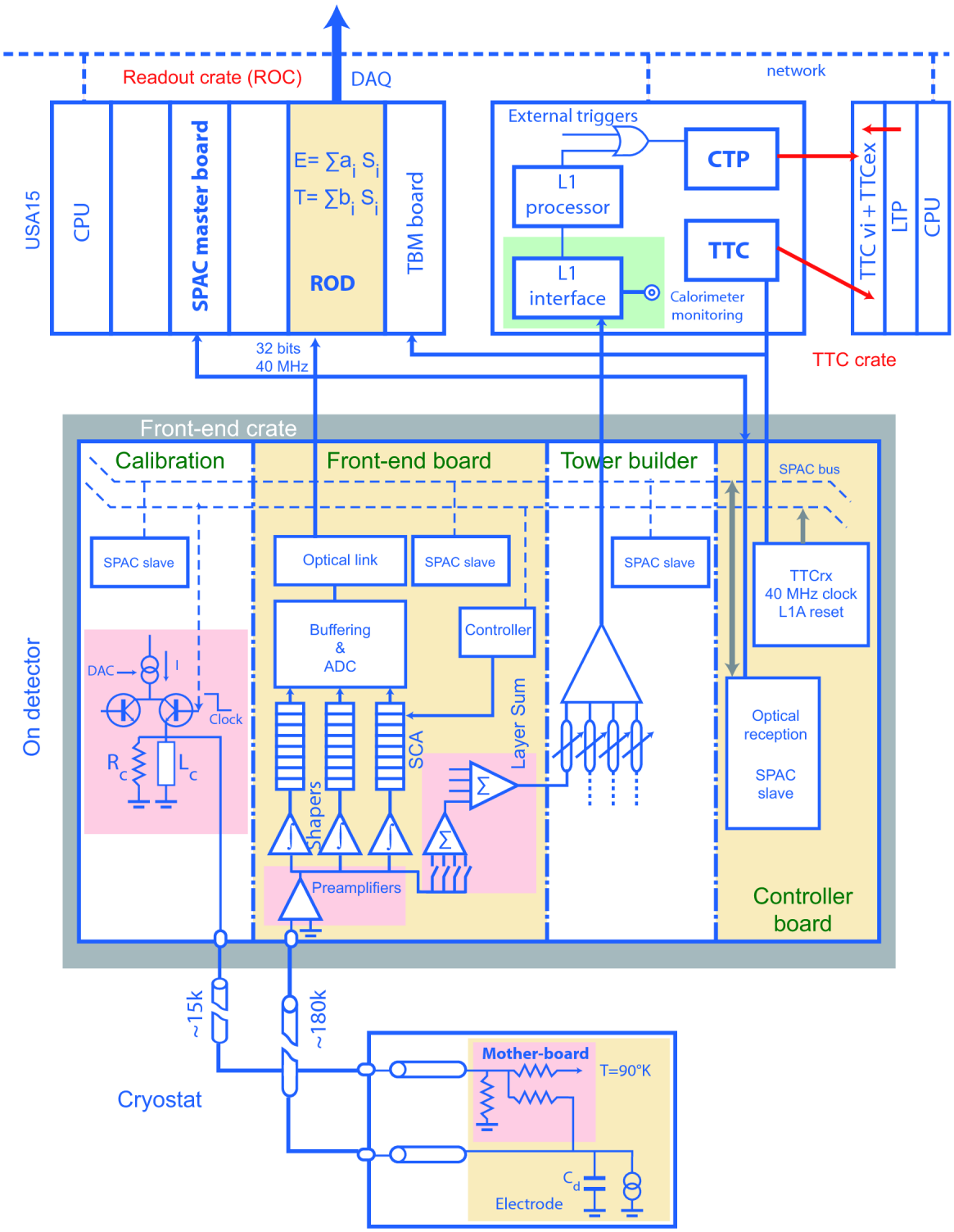


Figure 3.6: The current LAr calorimeter readout system. The LAr ionization signal proceeds upwards, through the front-end crates mounted on the detector and onto the back-end off-detector electronics. Diagram taken from [19].

also contain **Layer Sum Boards (LSBs)** that perform an analogue signal sum by layer. This sum is then sent to the **Tower Builder Board (TBB)** where all pre-summed signals from the LSBs of all calorimeter layers are added up to build a Trigger Tower. The total analogue sum is then provided to the L1 trigger. The signal processing chain in the FEB is shown in Figure 3.7

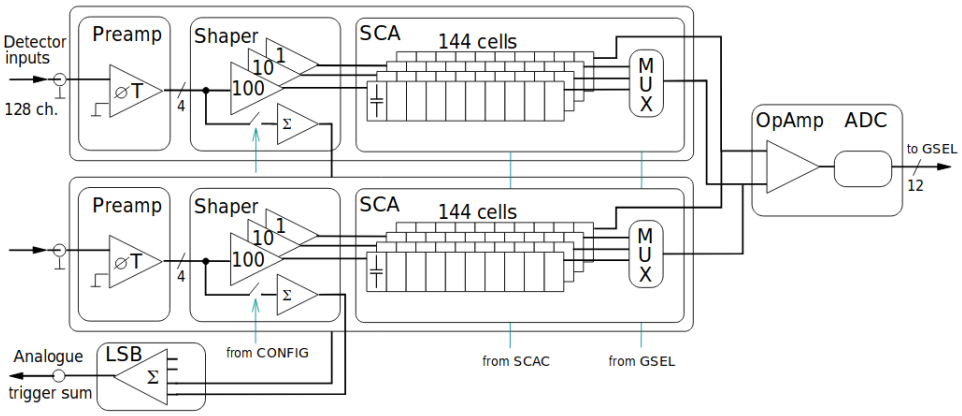


Figure 3.7: The analog signal processing chain of the LAr calorimeter FEB illustrating the pre-amplification, shaping, buffering and digitization. Diagram taken from [25].

Front end Processing chain

Raw signals from the ATLAS detector cells which are sent to the **FEB** have the typical triangular shape pulse as shown in Figure 3.8. The pulse has a sharp rise of a few nanoseconds and then a uniform falling edge up until 400 ns due to the drifting electrons in the detector cells.

As a first stage of amplification, the preamplifier serves to increase the current signal from a nA range to a mA range. The amplification is about 3.

The next stage is a CR-$(RC)^2$ shaper circuit, which is used to further optimise the signal-to-noise ratio and to remove the long tail of the detector response. The shaping circuit operates at a time constant $\tau = RC = 13$ns, representing a compromise between minimizing thermal noise, which decreases for slower shaping, and pile-up noise, which increases for slower shaping.

The result of the shaping is a bipolar, zero-integrated pulse that has a peaking time of about 40 ns as shown if Figure 3.8. The shaped signals are then split into three linear gain scales in order to achieve the full required dynamic range. The

absolute gain values are 0.8 (LO gain), 8.4 (MED gain), and 82 (HI gain).

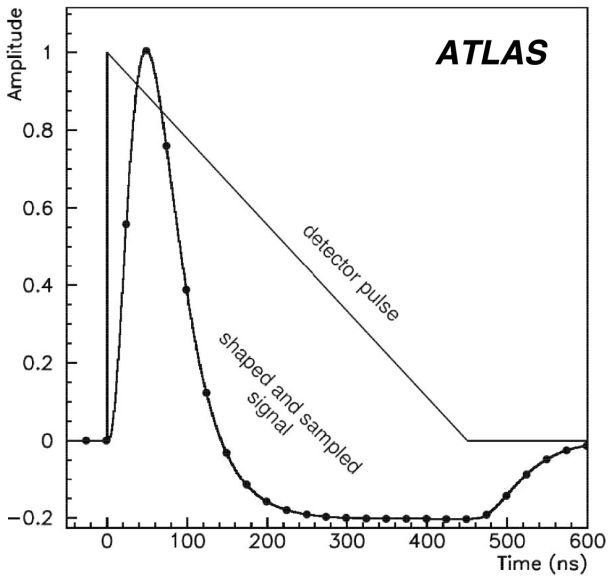


Figure 3.8: The triangular pulse generated in the detector is reshaped to a bipolar, zero-integrated pulse. It is sampled every 25 ns, indicated by dots on the diagram. Illustration taken from [24].

The shaped signals are then sampled at the LHC bunch crossing frequency of 40 MHz by **Switched-Capacitor Array (SCA)** analog pipeline chips. All three gain scales for each of four calorimeter channels are processed by the SCA. While data is being buffered in SCAs, a control element called the Switched-Capacitor Array Controller (SCAC) **ASIC** is used to manage the read and write addresses of the SCA such that non-triggered events are buffered awaiting the L1 trigger decision. Positively triggered events are then transferred to a de-randomising buffer of signals prior to digitisation.

At the end of the SCA, dual op amp chips are used to couple them to commercial 12-bit analogue-to-digital converter **ADCs** operating continuously at 40 MHz. The digitized data is formatted, multiplexed, serialized, and then transmitted via a 1.6 Gbps fibre optic link to the off-detector back-end electronics.

Back-end Processing chain

Each front-end crate is associated to a **Readout Driver (ROD)** crate, which performs digital filtering, formatting, and monitoring of the calorimeter signals. Each ROD receives digitized samples from up to eight FEBs and thus processes up to 1024

detector cells.

A ROD holds four processing units, each with 2 **DSPs** which can perform 5.7×10^9 instructions per second. The calculation done by the DSPs is an optimal filtering algorithm used to calculate the energy deposited in the calorimeter from the digitized samples. For deposits above a given (programmable) energy threshold, the time of the energy deposition, and the quality of the pulse are also calculated. The filtered energy, the signal time, and a pulse quality factor are then finally forwarded to the **DAQ**.

Data Acquisition System

The main task for the DAQ is to receive the filtered, digitized data from the ROD back end electronics. If the Level 1 trigger accepts an event, the event data is transferred to detector-specific RODs. If the event passes the HLT as well, the event data is transferred to permanent storage at the CERN computer centre.

Chapter 4

HL-LHC and LAr Upgrades

The current operating parameters of the **LHC**, luminosity of $\mathcal{L} = 1 \times 10^{34}\text{cm}^{-2}\text{s}^{-1}$, and a center of mass energy of $\sqrt{s}$ =13 TeV were obtained through an upgrade carried out in 2013/2014 known as **Long Shut Down 1 (LS1)**. **ATLAS** has recorded 139 fb^{-1} of data with these parameters.

With the current machine capabilities, ATLAS discovered the Higgs Boson, one of the major experimental feats of the century. However, to probe further the properties of the Higgs Boson, more sensitive measurements are required. For example, to measure various rare Higgs Decays (e.g. $H \to \tau + \tau$) as well as to improve measurements on already measured decays (e.g. $H \to Z + Z*$), a very good trigger performance is required to identify the final states of the leptons and jets.

There are also several motivations for collecting more data other than Higgs measurements. In particular, signatures for Beyond the Standard Model (BSM) Physics as well as flavour physics predict very rare processes and thus require large experimental luminosities for their detections.

A series of dedicated upgrades is thus planned to oversee the transition from LHC to the **HL-LHC**. The HL-LHC is planned to work at $\sqrt{s}$ =14 TeV and a peak luminosity of $\mathcal{L} = 7 \times 10^{34}\text{cm}^{-2}\text{s}^{-1}$. A timeline, consisting of the upgrades, is illustrated in Figure 4.1. The current status of the LHC is in **Phase 1**, an intermediate stage that would eventually lead to the realization of the HL-LHC. The following sections highlight the planned upgrades with a focus on changes relating to the ATLAS LAr Calorimeter.

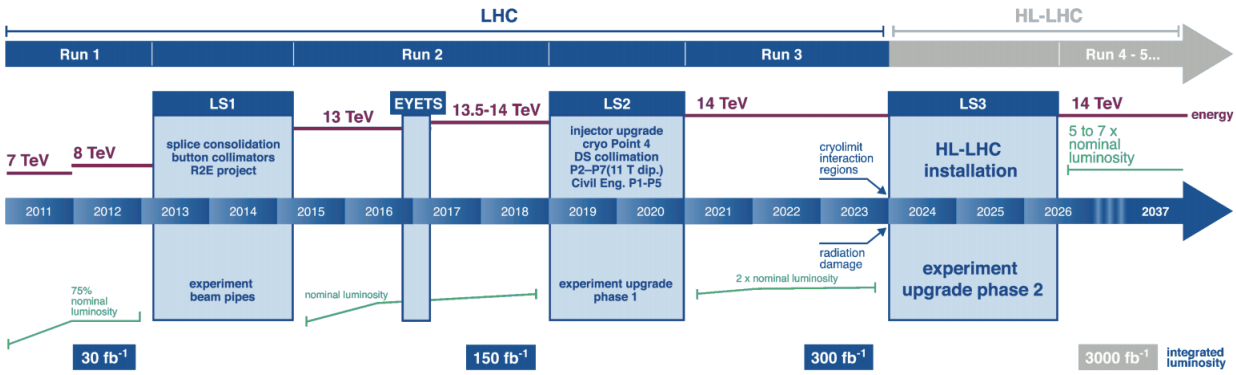


Figure 4.1: LHC upgrades baseline plan. Taken from [24].

4.1 Phase I

Phase I forsees a luminosity increase to $\mathcal{L} = 2 \times 10^{34}\text{cm}^{-2}\text{s}^{-1}$. The bunch crossing frequency remains the same at 40 MHz. The increase in luminosity would see the mean number of interactions per bunch crossing increasing to $\mu = 60$. The new electronics are thus tasked to keep the trigger threshold low to increase signal acceptance.

In an attempt to increase trigger information, higher segmentation of the Electromagnetic calorimeter is introduced. The Phase I upgrades replace the coarse trigger-towers to a much finer architecture using so called "Super Cells".

The result of the granularity increase is shown in Figure 4.2, which compares the energy deposition of an electron prior to and post LS2 (Phase I). The higher trigger energy resolution allows differentiation between leptonic showers and jets to be applied much sooner at the L1 trigger level. Further, this resolution helps enhance discrimination against backgrounds and fakes in a high-luminosity environment.

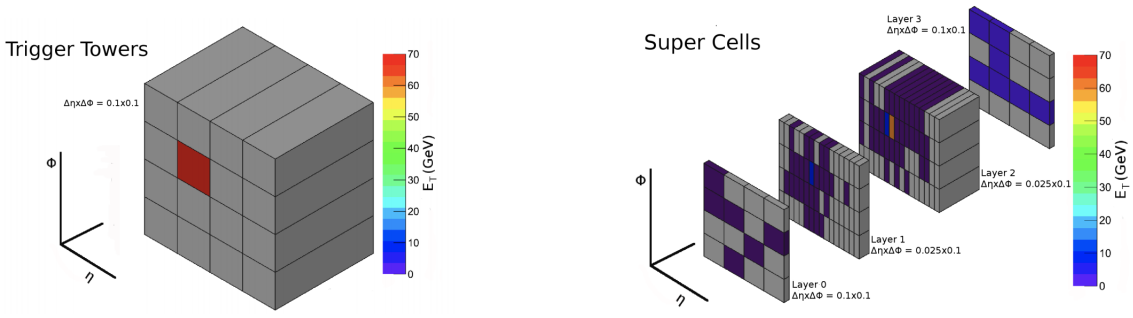


Figure 4.2: An electron (with 70 GeV of transverse energy) as seen by (a) the existing L1 Calorimeter trigger electronics and (b) Phase-I trigger electronics. Diagram taken from [26].

The geometric construction now provides information for each calorimeter layer for the full η range of the calorimeter, and as mentioned, finer segmentation in the front and middle layers of the EMB and EMEC. This is shown in Figure 4.3.

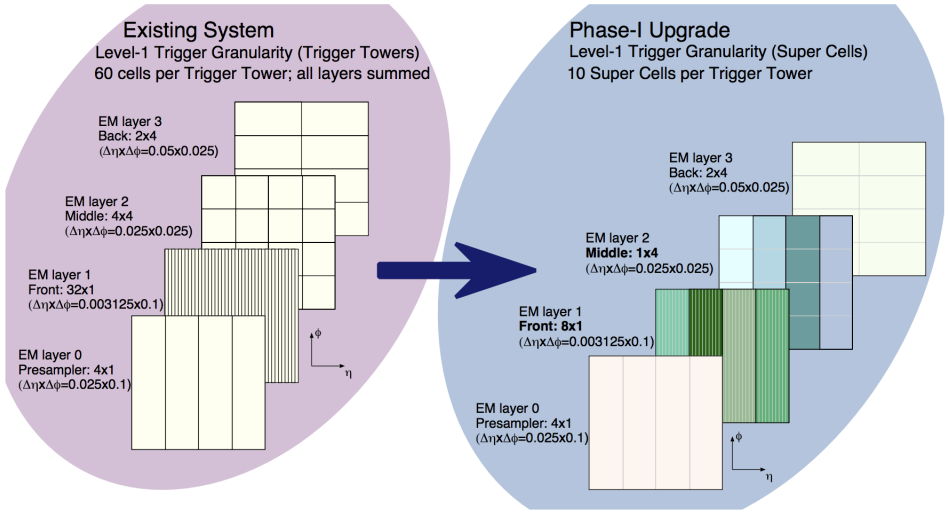


Figure 4.3: Geometric representation of the Trigger tower (prior to Phase I), where the transverse energies in all four layers are summed and the new Super Cells (post Phase I), where the transverse energy has finer granularity in the front and middle layers. Diagram taken from [26].

The supercells are realised by constructing and installing new **LSBs** on the **FEBs**. The FEBs will now consist of the existing system as well as the new LSBs working concurrently together. Using Super cells allows for more detailed reconstruction algorithms to be applied to make use of the higher η granularity. This higher granularity allows differentiation between leptonic showers and jets to be applied much sooner at the L1 trigger level.

The Phase I upgrade implements the use of a new **LAr Trigger Digitizer Board (LTDB)**, whose first purpose is to digitize the Super-Cell signals continually at 40 MHz. An added task of the LTDB is to build analogue sums that feed the legacy **TBB** to ensure a smooth transition while maintaining compatibility with the current system. The TBBs will eventually be retired during LS3.

Changes in the back end electronics are also implemented, introducing a new **LAr digital processing system (LDPS)**. The LTDB board serializes the digitized signals and sends them to the back-end LDPS via optical links (discussed further in Phase II). The LDPS is responsible for energy construction providing digitized measurements of the energies deposited in the calorimeter cells to the newly created

Feature Extractor (FEX) units in the L1 trigger system.

4.2 Phase II

Phase II upgrades are designed to prepare for the **HL-LHC**, operating at ∼7 times more luminosity than its current state. The $\sqrt{s}$ is not changed from Run 3 (kept at 14 TeV). Because of the increase in luminosity, pileup up to $\langle\mu\rangle = 200$ is expected.

The HL-LHC accelerator parameters require some changes to the front end electronics. The radiation limit of the current electronics is set to be exceeded during run 4. Because of this, new components with a larger radiation tolerance are to be installed in this upgrade. Another reason for the changes in the electronics is to obtain a more discriminating trigger that can perform effectively in the higher **pileup** environment. To do this, newer systems using robust architectures capable of performing sophisticated techniques will be installed in this upgrade.

Phase II upgrades will consist of systems that digitize each of the 182468 calorimeter readout channels at 40 MHz. To be able to do this effectively, both front-end and back-end (off-detector) systems are upgraded. Figure 4.4 illustrates the schematic of the final system after the Phase II upgrade. Certain elements such as the **LTDB** and the **LDPS** will already be installed in the Phase-I upgrade and remain operational in the HL-LHC phase to provide **Super Cell** information to the trigger system.

The following sections give an overview of the main systems that relate to the work of this thesis.

4.2.1 Front End Boards

New readout **Front-End boards (FEB2)** will be installed on the detector. These are different from the legacy Front-End boards (Figure 3.7) in that the **SCAs** are no longer needed as digitisation is done directly after shaping without any delay. A summary of the sequential functions of the FEB2 is shown in Figure 4.5 and is further elaborated below.

Each FEB2 receives the signal from 128 calorimeter cells, thus requiring a total of 1524 FEB2 boards to read out the entire LAr system. Two input connectors bring 64 calorimeter signals each from the crate baseplane to the FEB2 board.

Analogue processing is then done on the signal which includes pre-amplification, splitting into two overlapping linear gain scales, and shaping. All these processes

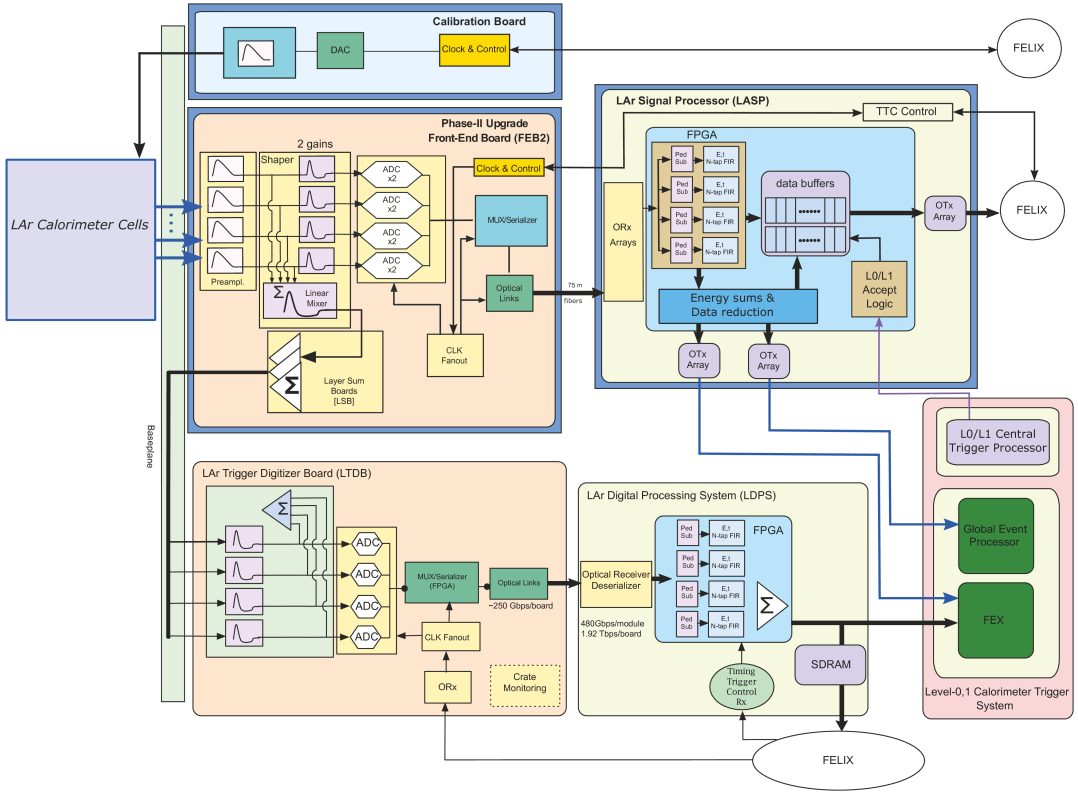


Figure 4.4: Schematic block diagram of the LAr calorimeter readout architecture for the Phase-II upgrade. The FEB2 board and the LASP will be the new components installed for this upgrade, while the LTDB and the LPDS will be inherited from Phase-I upgrade. Diagram taken from [24].

are implemented by a single **ASIC**. The shaping adopts a similar CR-$(RC)^2$ solution as with the current system but with the shaping time adjustable to account for the increased pileup expected at the HL-LHC.

The output of the preamp/shaper is connected to a 14-bit **ADC** that digitizes both of the two gain scale outputs for four channels each. Using the two gain system enables coverage of the full 16-bit dynamic range. The 16-bit dynamic range is chosen so that the ADC's least significant bit (LSB) value is less than LAr cell's electronic noise.

The ADCs operate at 40 MHz, synchronized to the LHC machine clock and bunch crossing frequency. **Bunch Crossing Identification (BCID)** information will be provided for each ADC chip to guarantee the correct synchronization of the calorimeter data. As with the current system, at every LHC orbit, a **Bunch Counter Reset (BCR)** signal will be distributed. Once the BCR signal is received on each FEB2 board, the BCID counters will start to generate the appropriate BCID information. The BCID is then added to the output of the ADC. Each ADC has 8 outputs each of which has a 14-bit ADC signal formatted into a 16-bit word and 16-bit word of BCID. The ADC outputs these words serially at a bit rate of 640 Mbps.

Serializer chips then receive 14 streams of ADC outputs and serialize the digital data with a standard protocol into a single bit-stream at 10.24 Gbps. This is done using lpGBT chips. **lpGBT (Low Power GigaBit Transceivers)** are custom CERN-made ASIC chips implemented using 65 nm-**CMOS** radiation tolerant technology. The lpGBT chips offer data transmission with fixed and "deterministic" latency for two-way communication, known as uplink and downlink. The lpGBT chips also offer encoding/decoding schemes and error recovery checks. The lpGBT can transmit uplink (from FEB2 to the **LASP**) data streams at 10.24 Gbps and can receive downlink (from LASP to FEB2) at 2.56 Gbps. Each lpGBT has 14 input channels (known as ePorts). To accommodate the 8 output streams from the ADC, a mapping scheme is designed such that 2 lpGBT chips serve 3 ADC chips, with 22 lpGBT chips per FEB2 board [24].

Finally, the serial lpGBT outputs are then converted from electrical to optical signals using **Versatile Link+ (VL+)** [27] and are then transmitted off detector.

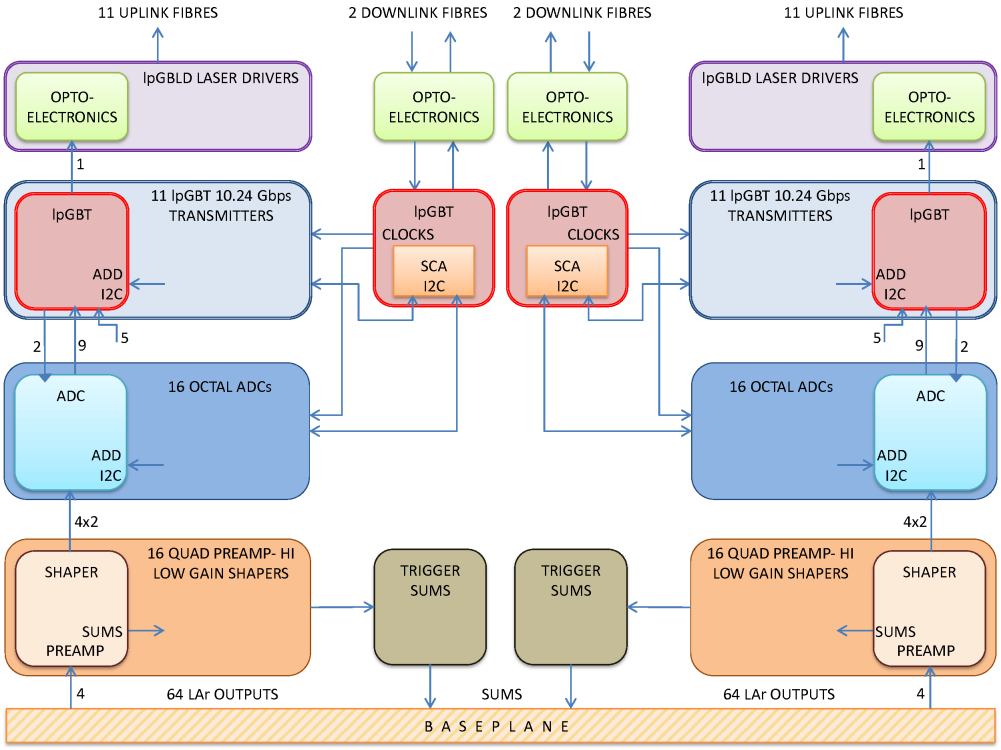


Figure 4.5: Overall architecture of the FEB2 board. 2×11 uplink fibres and 2× 2 downlink fibres are transmitted and received to and from the LASP, respectively. Diagram taken from [24].

4.2.2 Off-detector Electronics

For the purposes of this thesis, the major component concerned with the off-detector electronics is the **Liquid Argon Signal Processing (LASP)** system. The LASP is responsible for processing the digitized waveforms and transmitting the relevant information to the trigger and **DAQ** systems. The LASP hardware implementation will be an evolution of the **LDPS** which is installed in Phase I of the upgrades. The data stream digitized at 40 MHz from the FEB2 will be available for the LASP modules to perform a determination of calibrated cell energies and of signal times with respect to the bunch crossing time including an active correction of out-of-time pileup [24].

Thus, the LASP units require a high input and output bandwidth and flexible programming of data handling, digital filtering, and data reduction algorithms [24]. Many of the algorithms used by the LASP will be refined and optimized once the HL-LHC is running and operational knowledge is gained. For this reason, the LASP is realised with **Field Programmable Gate Arrays (FPGAs)** - a technology

that allows flexibility in re-programming hardware, as the main processing units [28]. Current prototypes of the LASP are based upon the Intel Stratix 10 FPGA Board (ref. Chapter 5.3). Two LASP FPGAs are mounted on each LASP unit (so-called LASP blade).

Interface with the LAr front-end electronics

The FEB2 connects with the LASP via optical links driven by **lpGBT** chips and **VL+** optical converters. Each LASP FPGA receives the raw data from 4 FEB2s, which means that 380 LASP FPGAs (or 190 LASP blades) will be required for the whole calorimeter readout [29].

The 22 output fibres from each FEB2 board are received by the LASP using the lpGBT protocol and FEC5 error correction [24]. Each link provides 224 user bits per 256 bit frame transmitted at 40 MHz, i.e. a user bandwidth of 8.96 Gbps out of 10.24 Gbps total.

Two more fibre pairs per FEB2 are also added to be sent to the LASP for **Trigger, Timing and Control (TTC)** for clock distribution, control, and monitoring purposes.

Modules of the LASP

The LASP has been subdivided into modules responsible for doing certain tasks. An overview schematic of the proposed modules in the LASP is shown in Figure 4.6

The crucial modules in the main data path and their functions are listed below [29]:

- *lolli* - controls the hardware components of the LASP and is integrated in the FPGA;
- *ialign* - deserializes, demultiplexes and aligns in time the ADC data from the FEB2;
- *remap* - remaps incoming data following the detector geometry, in a configurable way;
- *dacore* - applies a filter algorithm for energy reconstruction and Bunch Crossing Identification (BCID) for the data and trigger path;
- *buffs* - buffers raw, reconstructed and data sent to the trigger;

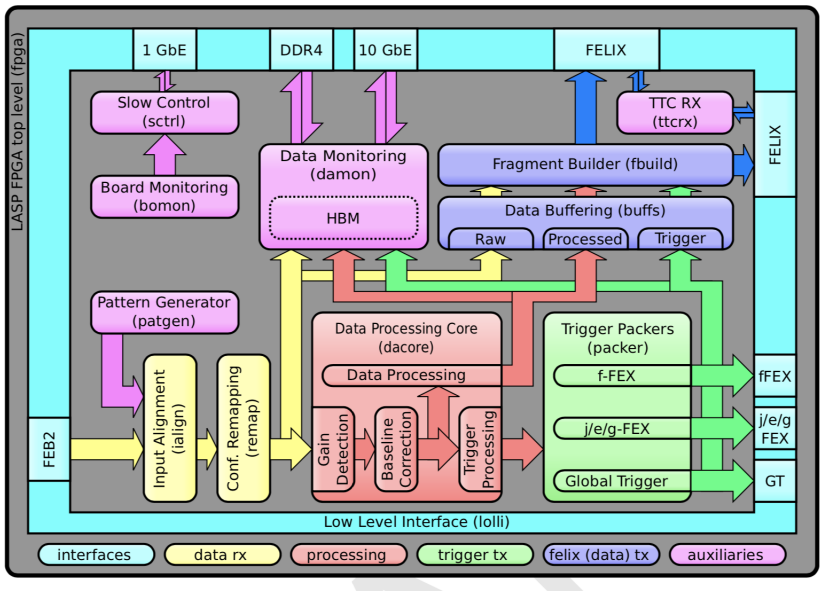


Figure 4.6: LASP FPGA firmware block diagram with the main blocks and interfaces. Diagram taken from [29].

- *fbuild* - builds data packets for triggered events and provides them the the DAQ system via FELIX [30].

In addition, the following functional components are also part of the LASP:

- *packer* - makes the proper geometrical sums and formats the data in the trigger data path and provides them to the FEXs and Global Trigger;
- *ttcrx*: decodes and provides signals from the TTC system via FELIX;
- *damon* - carries out the monitoring of data and their transfer to DAQ;
- *bomon* - carries out the monitoring of the LASP itself (status, temperature) to the ATLAS Detector Control System (DCS);
- *sctrl* - serves as the slow-control system to configure and monitor the LASP FPGA and all its functions;
- *patgen* - serves an optional alternative to the data from the FEB2s and injects user defined data.

Data flow to the Trigger and to the DAQ

The LASPs directly feed the ATLAS trigger system: For each BC the LASPs will send the corresponding energies, packed as 10-bit values to the Global Trigger [29]. The Interface of the LASPs to the **DAQ** system is provided by the **FELIX (FrontEnd LInk eXchange)** network interface modules which are planned to receive and send data with a lpGBT compatible link speed.

Measurements using the digitized signal such as precision energy values, the peaking time of signal pulses, and pulse quality parameters are calculated for each Bunch crossing. These data are stored until the Level 0 trigger accept (L0A) or Level 1 trigger accept (L1A) signals arrive, which are received via FELIX boards as part of the **TTC** system. The buffering time is up to 35 μs.

Moreover, all the information regarding calculated energies, signal time, and signal quality parameters that are sent to the trigger system are stored in memory until a trigger accept. With the reception of L0A/L1A, the precision and trigger data are sent to the DAQ system, again via the FELIX boards [29].

Chapter 5

Injector Project Design

Part of the development in the **Phase II LAr LASP** Process involves verification and testing of modules to ensure correct functionality. For this reason, a firmware implementation of a data injector is designed to support the needs of the Phase II LASP upgrade. The motivations behind a data injector and the benefits obtained from this module will be given in Section 5.1. The design specifications required by the model are presented in Section 5.2. Important technical information on **FPGAs** is provided in Section 5.3 as a means to better understand the firmware as well as the tools in which the Injector project is developed. An overview of a Monte-Carlo simulation program that mimics **FEB2** pulses at $\mu = 200$ is given in Section 5.4

5.1 Introduction and Motivation

Whilst the development of the LASP is taking place, the LHC will be either in Run 3 ($\int \mathcal{L} = 300\text{fb}^{-1}$) or in **Long Shut-Down 3 (LS3)**. During either of these phases, the new Front-End Board planned for the Phase II upgrade (FEB2) will not be operational. Thus, the input to the LASP will not be available until full integration and completion of the Phase II upgrade.

The lack of input to the LASP presents a problem within its development cycle as there is no means to verify the proper functionality of individual LASP modules.

Data into the LASP prototypes that mimics the data expected from the FEB2 is required to test the firmware code and hardware as the LASP is being developed. Therefore, the Injector project is a crucial component in the LASP Firmware Work-packages framework, specifically in *WP5: Firmware Specification* and *WP7:*

Firmware Integration [28].

The Injector project is especially vital for testing during the pre-prototype and prototype stages of the LASP. The ability to test scenarios for both regular operation (expected or nominal inputs) and edge cases and possible failure modes (injection of corrupted data, bad signal timing, disrespecting input protocols) is a feature that makes the Injector project so important. The results from the validation of the data Injector tests will inform the hardware design and specifications needed for the actual implementation of the LASP.

As an integral part of the LASP test-bench, the data injector will evolve as the LASP design progresses. This includes adding further functionality including increasing length of time data can be injected as well as quantity of data that can be injected. The injector can also evolve by adding more complexity by using additional hardware such as the **FrontEnd LInk eXchange (FELIX)** as it becomes available.

5.2 General specifications

In order to be useful as a test-bench, the Injector project must inject data at a rate and quantity that mimics that of the FEB2 during actual operation. These two quantities set the design constraints that the Injector project must meet.

5.2.1 Quantity of Data

In the baseline design, one LASP FPGA unit receives data streams from 4 FEB2s (Figure 4.5). The data content of each of signals transmitted by the FEB2 consists of a 256-bit wide payload (information in transmitted data), 230 of which are user-controlled and the other 26 bits of which are error correction and frame aligner information. This is shown in Figure 5.1

The 230 user-controlled bits consist of twelve 16-bit **ADC** words and two 16-bit **BCID** bits with six unused bits. The 16-bit ADC words are made from two 14-bit ADCs, one ADC encoding high gain data, and the other ADC encoding low gain data. Thus, the actual signal from the calorimeter channels will be encapsulated in the $12\times16=192$ bits of the 230 user-controlled bits [24, 29].

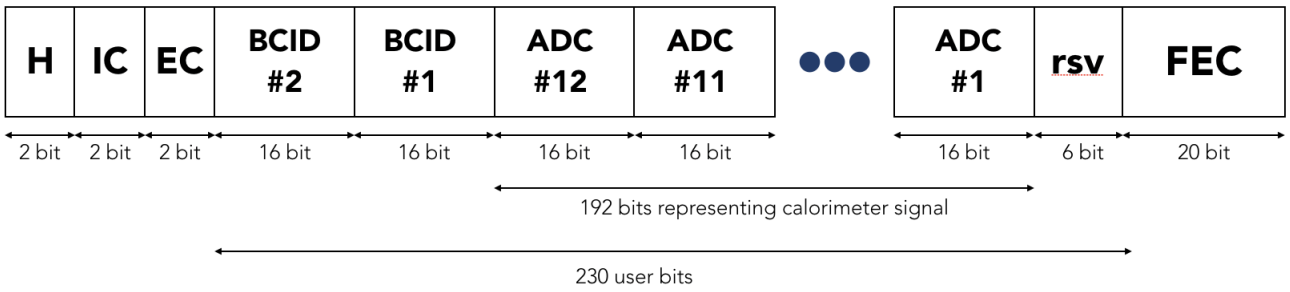


Figure 5.1: Contents of the lpGBT payload. H represents the header, IC represents Internal slow control bits, EC represents External slow Control bits, BCID represents the Bunch crossing of the signals contained within the payload, ADCs represents the signal obtained from the calorimeter channels, rsv represents reserved bits, FEC represents the Forward Error Correction bits

Critical Specification #1

Ignoring the presence of the other bits, the data injector must be able to provide 12×16-bit ADCs to the LASP for every payload.

5.2.2 Rate of Data

LAr electronics will read the calorimeter cells at the bunch crossing frequency of the LHC i.e. 40 MHz. This means that the readout will be updated every 25 ns.

The lpGBT payload in the FEB2 is serialized and 12 streams of the ADC output + 2 streams of the BCID are converted into a single bitstream at 10.24 Gbps. This is then converted into an optical signal and sent to the LASP.

The Injector project must thus replicate this and be able to inject the 230 user controlled bits + the 26 supplementary bits at 10.24 Gbps i.e. every 25 ns.

Critical Specification #2

Ignoring the presence of the other bits, the data injector must be able to provide different 230-user controlled bits every 25ns.

5.3 FPGAs and Stratix 10

5.3.1 FPGAs

A **Field Programmable Gate Array (FPGA)** is a highly integrated circuit logical device that contains two-dimensional arrays of generic logic cells and programmable switches. A logic cell can be programmed/configured to perform a simple logic function (i.e. AND, OR, NOT, XOR, etc), and a programmable switch can be customized to provide interconnections among the logic cells [31]. A simple visualization of this can be seen in figure 5.2. Users can implement a custom design by specifying the function of each logic cell and selectively setting the connection of each programmable switch.

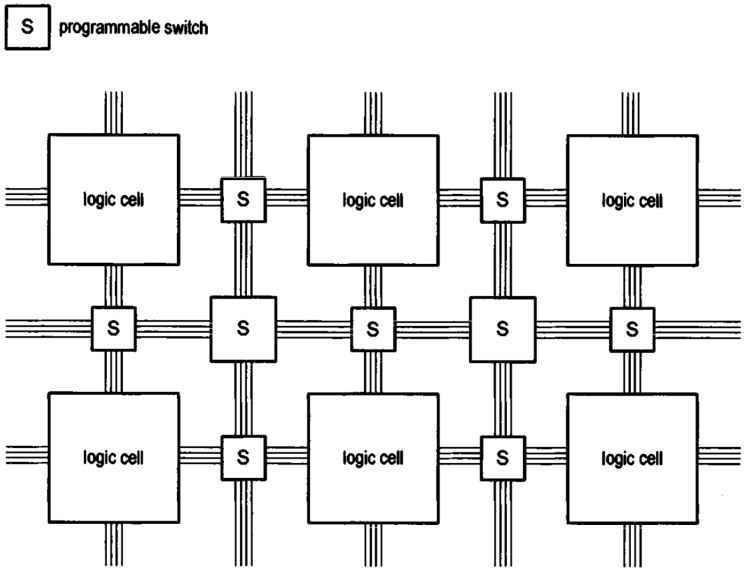


Figure 5.2: Conceptual structure showing the interconnects of an FPGA device. Diagram taken from [31].

FPGAs have nowadays found themselves in various applications. They are being used to replace **ASICs, Digital Signal Processors (DSP),**‘ Central Processing Units (CPU), and Graphics Processing Units (GPU). One of the reasons FPGAs are being used extensively, despite being clocked at lower frequencies, is due to their configurability, the massive parallelism, and the integrated processing blocks. These blocks typically provide the functionality of several DSPs at once and the logic elements can create more flexible combinations of processing steps [32]. This adds to the hardware flexibility of FPGAs.

FPGAs are configured by writing designs in a **Hardware Description Language (HDL)**. The configuration of an FPGA does not represent a program that is executed as it is the case for processors or microcontrollers. Instead, the HDL code design describes the structure and behaviour of electronic circuits on a higher level of abstraction. Thus it is possible to describe processes that run completely independently and in parallel to each other. A high speed of execution can be reached and the overall integrity is only limited by the logic resources of the FPGA used [33]. Once an HDL design has been completed, it can be synthesized using a programmable logic device design software (e.g. Intel **Quartus Prime**). Using a simple adaptor cable, the synthesized design can then be downloaded to the FPGA device to obtain the custom circuit.

To help make the FPGA devices user-friendly and to optimize device performance, FPGAs now contain **Lookup Table (LUT)** based logic cells as well as Macro cells.

LUT-based logic cells

A logic cell usually contains a small configurable combinational circuit with a D-type **flip-flop** (D FF). The most common method to implement a configurable combinational circuit is a LUT. An n-input LUT can be considered as a small 2^n-by-1 memory. The LUT can then be used to implement any n-input combinational function.

An example of a three-input LUT implementation of $a \oplus b \oplus c$ is shown in Figure 5.3.

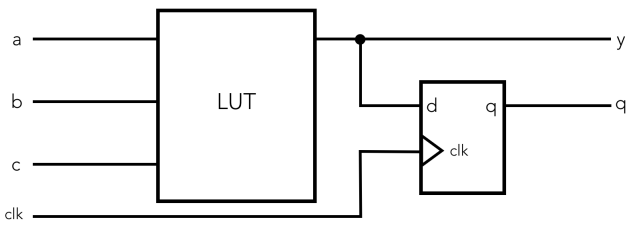


Figure 5.3: Conceptual diagram of the LUT Implementation

a	b	c	y
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Figure 5.4: LUT for $a \oplus b \oplus c$

Macro cells

Most FPGA devices embed certain **macro cells** or a group of macro cells called **macro blocks**. These are designed and fabricated at the transistor level, and their functionalities complement the general logic cells [31]. Macro cells can be used to provide memory blocks (a group of registers) for storage capacity in an FPGA, combinational multipliers which allow rapid processing of the expensive multiplication operator, clock management circuits which allow for derivation of a faster or slower **Phase-Locked Loop (PLL)** clock from a reference oscillator, and **Input/Output (I/O)** interface circuits. Some advanced FPGA devices may even contain one or more prefabricated processor cores built into the fabric of the FPGA. Such FPGAs are so called **System-On-a-Chip (SoC)**.

5.3.2 Intel Stratix 10 GX FPGA

The Intel **Stratix 10** FPGA is the latest variant of the Stratix family that uses 14 nm Tri-Gate transistor technology that was first released in 2017. The Stratix 10 sub-branch has many variants that are optimized on different FPGA features. The GX devices are the basic Stratix 10 release with 28.3 Gbps transceivers. The SX SoC devices also have 28.3 Gbps transceivers but additionally, have a 64 bit quad-core **ARM** A53 processor. The TX devices deliver the most advanced transceiver capabilities allowing a single transceiver channel to operate up to 56 Gbps in **PAM**-4 mode. The MX devices have incorporate 3D stacked high-bandwidth memory 2 (HBM2) in a single package which provides massive on-chip memory capacity.

For the development of the LASP prototype, Stratix 10 GX boards are being used [29]. The GX board has sufficient transceiver capacity with regards to the single bitstream that will be sent from the FEB2 to the LASP at 10.24 Gps. Compared to its other Intel counterparts (Arria, Cyclone), the Stratix family has a much wider feature range and a much higher performance capability. The latest commercially available variant of the Stratix family - the Stratix 10, was thus chosen as a candidate of the LASP [28]. An overview of the available features in the Stratix 10 device is shown in Figure 5.5. The physical Stratix 10 board is shown in Figure 5.6.

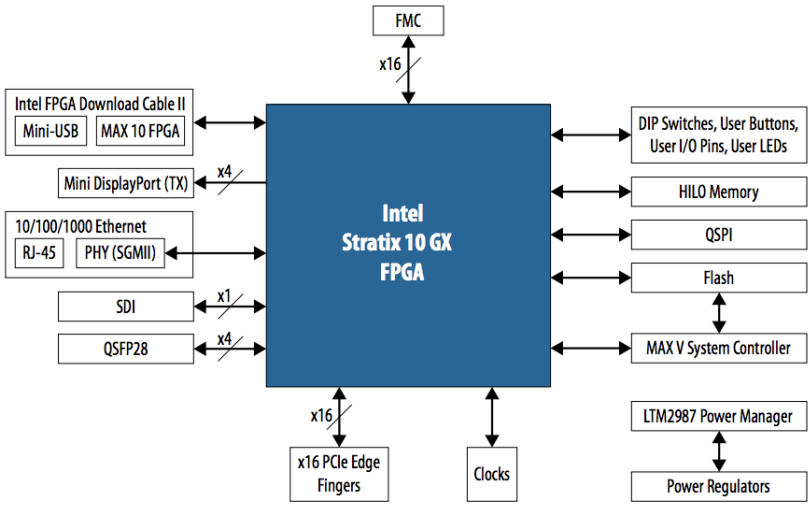


Figure 5.5: Block diagram of features supported by the Stratix 10 Device. Taken from [34].

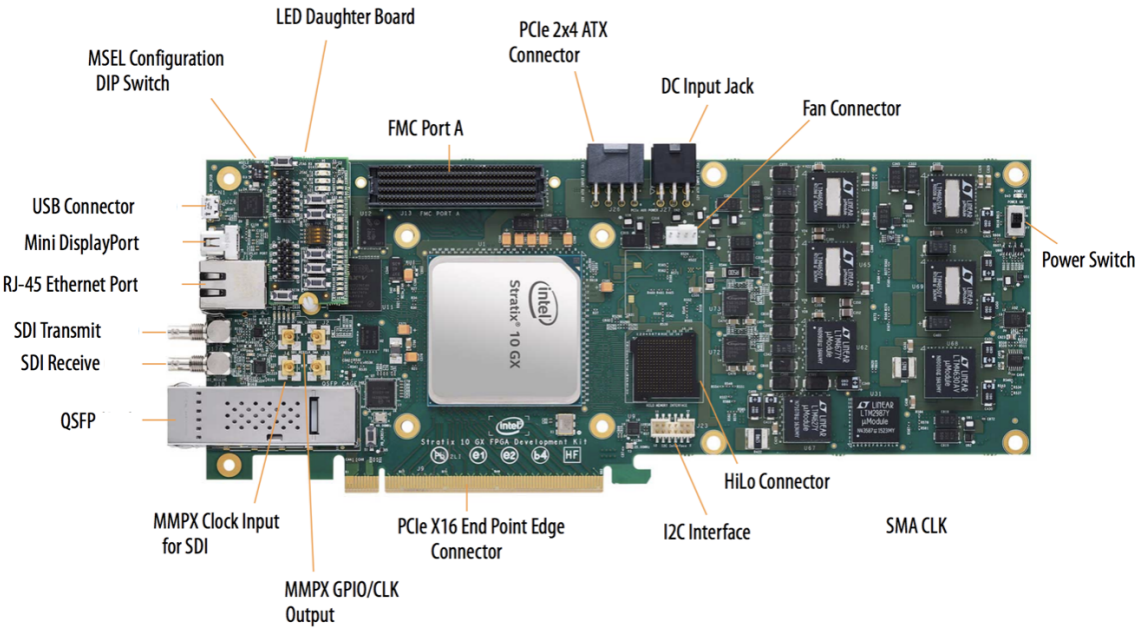


Figure 5.6: Frontal Image of the Stratix 10 GX FPGA Development kit. Taken from [34].

Logic cells

Logic cells in the Stratix 10 device are split between **Logical Elements (LEs)** and **Adaptive logic modules (ALMs)**. An LE contains a four-input **LUT**, which can implement any function of four variables, and a D **FF**. The more powerful ALMs

contain an eight-input fracturable LUT, two dedicated embedded adders, and four dedicated registers (D FFs). ALMs can be divided into two combinational adaptive LUTs each with four inputs. The ability to use ALMs as either four-input or eight-input LUTs makes it possible to implement various combinations of two functions or a subset of eight input functions. The adaptability also allows an ALM to be completely backward-compatible with four input LUT architectures. A block diagram of an ALM implemented in the Stratix 10 is shown in Figure 5.7.

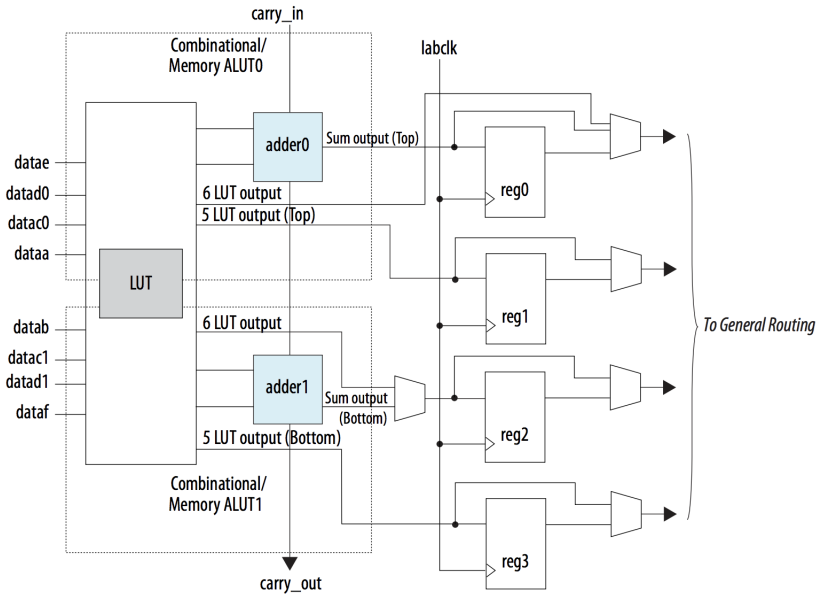


Figure 5.7: Intel Stratix 10 ALM High-Level Block Diagram. Diagram taken from [35].

Interconnect structure

As the size of the transistor shrinks, the routing delay becomes a significant portion of a circuit's overall propagation delay. To optimize performance, routing is performed on several levels. The conceptual diagram is shown in Figure 5.8. On the local level, 10 ALMs are grouped together to form a **LAB (logic array block)**. The ALMs within the same LAB are connected via the local interconnect and signals can be routed within the LAB directly. There are also direct link interconnects for routing signals between the adjacent ALMs. On the global level, the ALMs and macro cells are connected via a two-dimensional row- and column-based MultiTrack interconnect structure. The MultiTrack interconnect is made of dedicated fixed-length horizontal (row) and vertical (column) routing channels [31].

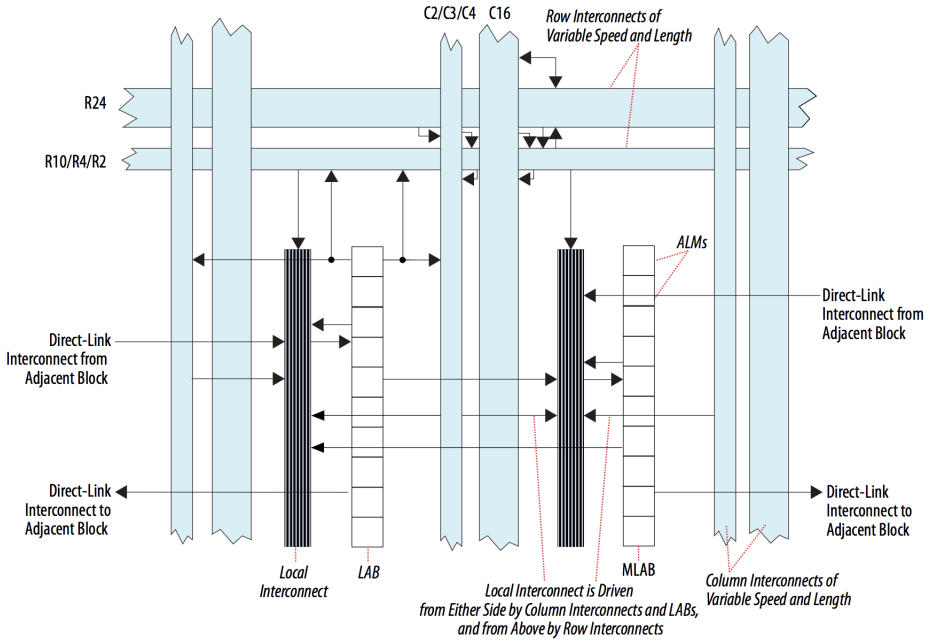


Figure 5.8: Intel Stratix 10 LAB Structure and Interconnects Overview. Diagram taken from [35].

Macro cells

The Stratix 10 have several types of **macro cells**, a few of which are: Embedded memory blocks, combinational multipliers, **DSP** blocks, Hard memory controller, and **PLLs**.

Embedded memory blocks are divided into Embedded **SRAM** (eSRAM) blocks, **M20K** blocks, and **memory logic array blocks (MLABs)**. An eSRAM block is a Fast path, low latency, high bandwidth, and very high random transaction rate on-chip memory block. A M20K block is a 20-kilobit memory storage block with dedicated resources making it ideal for storage of larger memory arrays. MLABs are adapted and made from 10 LABs which are optimized for wide and shallow memory arrays. This makes it a prime candidate for DSP applications and **FIFO** buffers.

The Hard Memory controller provides the capability to use an external memory which can be connected via the HiLo connector on the board. The Memory controller supports:

- DDR3 x72 @ 1067 MHz/2133 Mbps
- DDR4 x72 @ 1333 MHz/2666 Mbps

External memory devices can be helpful as they supplement the on-chip embedded memory blocks on the FPGA, which is often in the MB scale.

Resource Capability of Intel Stratix 10 GX FPGA

Resources	
Logic elements (LEs)	2,753,000
Adaptive logic modules (ALMs)	933,120
Registers	3,732,480
M20K memory blocks	11,721
M20K memory size (Mb)	229
MLAB memory size (Mb)	15
Transceiver Count	96
Variable-precision Digital Signal Processing (DSP) blocks	5,760
18 x 19 multipliers	11,520
Maximum user I/O pins	1,160
Memory devices supported	DDR3, DDR4, RLDRAM3

Table 5.1: Features of the Intel Stratix 10 GX 2800 FPGA Board. Adapted from [36].

5.3.3 NIOS II Soft-core processors

A soft-core processor is a microprocessor that can be wholly implemented using hardware logic synthesis. The soft-core processor designed for Intel FPGAs is called **NIOS II**. It is a part of Intel's **IP (Intellectual Property)** and is mapped onto the FPGA's logic and **macro cells**.

The NIOS II processor follows the basic design principles of a **RISC (Reduced Instruction Set Computer)** architecture and uses a small and optimized set of instructions. Its main components include thirty-two 32 bit general-purpose registers and control registers, an Arithmetic and Logic Unit, and an exception and interrupt handler. The NIOS II processor functions within a 32-bit address space

The NIOS II processor also utilizes separate ports for the instruction and data access. The instruction master port fetches instructions and performs read operations. The data master port reads data from memory in a load instruction and writes data to memory in a store instruction [31].

The NIOS II processor handles 32 level sensitive interrupt request inputs. When an exception or an interrupt occurs, the processor transfers the execution to a specific address. An interrupt service routine at this address then determines the cause of the interrupt and takes appropriate actions [31].

The NIOS II processor additionally has a **JTAG** Debug module that connects to signals inside the processor and can take control over the processor. The JTAG port can then be connected to a host PC to communicate with the debug module. This is the way NIOS programs are downloaded into memory. Other features of the debug module include setting breakpoints and examining register and memory contents [31].

5.3.4 Quartus Prime

Quartus Prime is Intel's version of programmable logic design software. It provides a design environment and user interface that supports compilation and programming for all Intel **FPGAs** and **SoC** devices.

Quartus management hierarchy starts with a project file (`.qpf`), which includes all relating logic and information for a particular project. Typical files that may be found in a project include:

- Project settings file (`.qsf`) - Lists design files, entity settings, target device, synthesis directives, and placement constraints;
- Timing constraints file (`.sdc`) - Contains clock properties, exceptions, setup/hold times;
- Logic Design File (Any **HDL** language such as `.vhdl`, `.v`) - Specifies high-level representations of a circuit from which lower-level representations and actual wiring can be derived;
- IP cores file (`.ip`) - Pre-designed modules that contain HDL codes for a particular function;
- Platform Designer files (`.qsys`) - A file that contains the interconnection of different **IPs**;
- Programming files (created post-compilation - `.sof`) - Contains device programming image and information that will be ultimately be loaded into the device.

Platform Designer - Quartus tools

Quartus has a certain set of tools that aid with the designing process. One such tool that is used extensively for the Injector project is the **Platform Designer.** Platform Designer is a system integration tool that simplifies the task of defining and integrating custom **IP** components (IP cores) into an FPGA design. The interconnect for many of the IPs is done using the Avalon Interface. Platform Designer automatically creates interconnect logic from high-level connectivity that the user specifies. The tool is popular in embedded design because it allows for faster development cycles by the automatic generation of interconnects with the HDL of the system. It also accounts for and automatically places peripherals to meet the timing requirements of a design.

Platform designer generates a `.qsys` file which describes the whole interconnection of the system as well as the individual `.ip` files.

Avalon Interface

The **Avalon Interface** is a standardized method to accommodate various communication needs and to connect components within an Intel FPGA chip. It can be thought of as a shared road within different components with each component occupying its own lane to avoid bottlenecks and communication conflicts. The Avalon standard consists of a few different interfaces such as Avalon Streaming, Avalon Memory Mapped and Avalon Conduit. The Injector project employs great use of the **Avalon memory-mapped interface (Avalon MM)**, which is treated below.

The Avalon-MM interface defines an address-based master-slave connection. An Avalon MM master uses an address to identify an Avalon MM slave and can initiate a transaction to read data from or write data to the slave [31]. Components are created to behave as either masters or slaves and this feature cannot be changed within a respective component IP.

The Avalon-MM interface defines several signals, a few which are listed below:

- `read` - Master → Slave: Asserted to indicate a read transfer;
- `write` - Master → Slave: Asserted to indicate a write transfer;
- `address` - Master → Slave: Used to specify an offset in the slave address space. Each value identifies a memory location in the slave address space;

- `readdata` - Slave → Master: Carries the data provided by a slave in a read operation;
- `readdatavalid` - Slave → Master: When asserted, indicates that the `readdata` signal contains valid data;
- `writedata` - Master → Slave: Carries the data provided by a master in a write operation;
- `waitrequest` - Slave → Master : A slave asserts this signal when unable to respond to a `read` or `write` request;
- `burstcount` - Master → Slave: Used to indicate the number of transfers in each burst.

An example of a sample read and write is shown in Figure 5.9. The master first asserts `address` and `read` on the rising edge of the clk 1. The `waitrequest` signal is asserted stalling the transfer. During this period, `address` and `read` signals are held constant. The slave then captures `readdata` on the rising edge of clk 3 and de-asserts `waitrequest`. The read transfer is then completed. `writedata`, `address`, and `write` signals are then asserted at the rising edge of clk 5. This causes `waitrequest` to stall. At the rising edge of clk 8, the slave captures `writedata` and de-asserts `waitrequest`, ending the transfer [37].

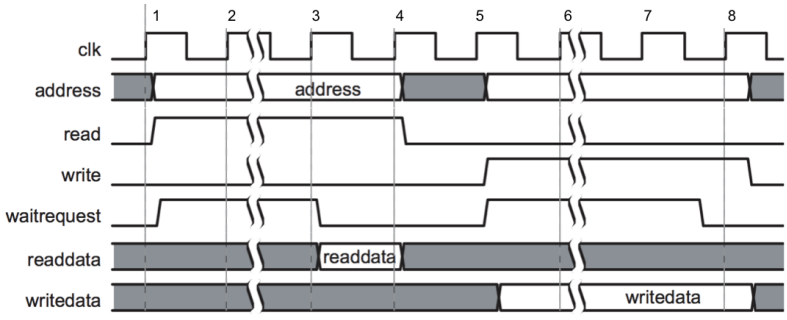


Figure 5.9: Slave Read and Write Transfer with Fixed Wait-States. Diagram taken from [37].

The Injector project also uses burst reads: a capability to perform multiple transfers as a unit, rather than treating every word independently. Bursts usually increase throughput for slave ports that achieve greater efficiency when handling multiple

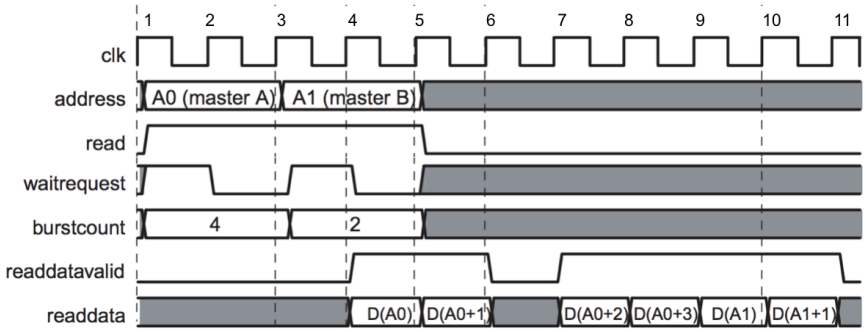


Figure 5.10: Slave Read Burst. Diagram taken from [37].

words at a time, such as for a DDR SDRAM. Figure 5.10 shows a bursting read transfer process.

Master A asserts `address (A0)`, `burstcount`, and `read` after the rising edge of clk 1. The slave asserts `waitrequest`, causing all inputs to be held constant through another clock cycle. The slave captures A0 and `burstcount` at this rising edge of clk 3. A new transfer could start on the next cycle. Master B drives `address (A1)`, `burstcount`, and `read`. The slave asserts `waitrequest`, causing all inputs to be held constant. At the rising edge of clk 4, the slave presents valid `readdata` and asserts `readdatavalid`, transferring the first word of data for master A. On the rising edge of clk 5, the second word for master A is transferred. The slave de-asserts `readdatavalid` pausing the read burst. The slave port can keep `readdatavalid` de-asserted for an arbitrary number of clock cycles. At the rising edge of clk 9, the first word for master B is returned [37].

Quartus Compilation

Given a set of logic and constraints, Quartus Prime compiles the files to report and create the programmable image (`.sof`) which can then be downloaded into an Intel FPGA board. The full compilation process is dependent on the FPGA device. The compilation process [38] includes:

1. Analysis & Synthesis - The analysis sub-stage checks for design file and project errors. The synthesis sub-stage synthesizes, optimizes, minimizes, and maps design logic to device resources.

2. Fitter

(a) Plan — places all periphery elements (such as I/Os and PLLs) and determines a legal clock plan.

(b) Place — places all elements in a legal location.

(c) Route — creates all routing between the elements in the design.

(d) Retime — moves (retimes) existing registers into Hyper-Registers for fine-grained performance improvement.

(e) Fitter (Finalize) - performs post-Route fix-up to correct any short path hold violations remaining from retiming.

3. Timing Analysis - Analyzes and validates the timing performance of all design logic with a Quartus tool called Timing Analyzer.

4. Assembler - Converts the Fitter's placement and routing assignments into a programming image for the FPGA device.

5.4 AREUS

In order to simulate the **FEB2** pulses, a Monte-Carlo simulation called **ATLAS Readout Electronics Upgrade Simulation (AREUS)** is used. AREUS was originally developed for the simulation of the full LAr trigger readout chain for the **Phase-I** upgrade. It is a fast, flexible, and full-featured detector simulation framework that is specially designed for the evaluation of the performance of all kinds of digital signal filters [39].

In AREUS, the simulated signals from a Monte-Carlo event generator are processed over a given time period on a **BC-by-BC basis**. The events are thus processed as they occur in the continuous data flow and the filter algorithms applied operate as in real time [40].

AREUS has the capability of taking into account the granularity of the LAr calorimeter cells, including the newly installed **super-cells**. The simulation software can generate pulses for elementary cells and also ranges of η and ϕ sections in the LAr calorimeter. Calorimeter cells are fed an input signal consistent with the generation of ATLAS events and AREUS reproduces the pulse shapes as consistently as possible with the electronic readout circuitry and within the LHC **pile-up** conditions specified. Hence, AREUS can be used to fully simulate the pulses expected from the FEB2 without the need of an actual hardware implementation.

Chapter 6

Injector Project Implementation

The Injector project is designed on an Intel **Stratix 10** GX **FPGA** Development kit (devkit) board, which uses technology similar to that for the **LASP**. The architecture for the project is broken down into three different sub-modules:

1. Data Injection - Sending data from a workstation to the FPGA . Further details of this are described in Section 6.2;

2. Data Storage - Storing the received data in an off-chip **RAM**. This sub-module is described in Section 6.3;

3. Data retrieval and transfer - Fetching of the stored data in the off-chip RAM and sending it to the LASP. This sub-module is described in Section 6.5.

A general overview of the whole project is given in Section 6.1. A full description of how the Injector Hardware and Software interfaces with each other is given in Section 6.4.

6.1 Overview of Design

A "bird's-eye view" of the overall design is shown in Figure 6.1, where the Injector project, implemented by the middle FPGA, is fed data from a workstation and injects data into the LASP.

AREUS simulations of **FEB2** pulses are performed on and stored in a workstation. To enable the transfer of this data to the Injector-configured FPGA, two options are available: (i) **JTAG UART** and (ii) **Ethernet** protocol. The JTAG UART is

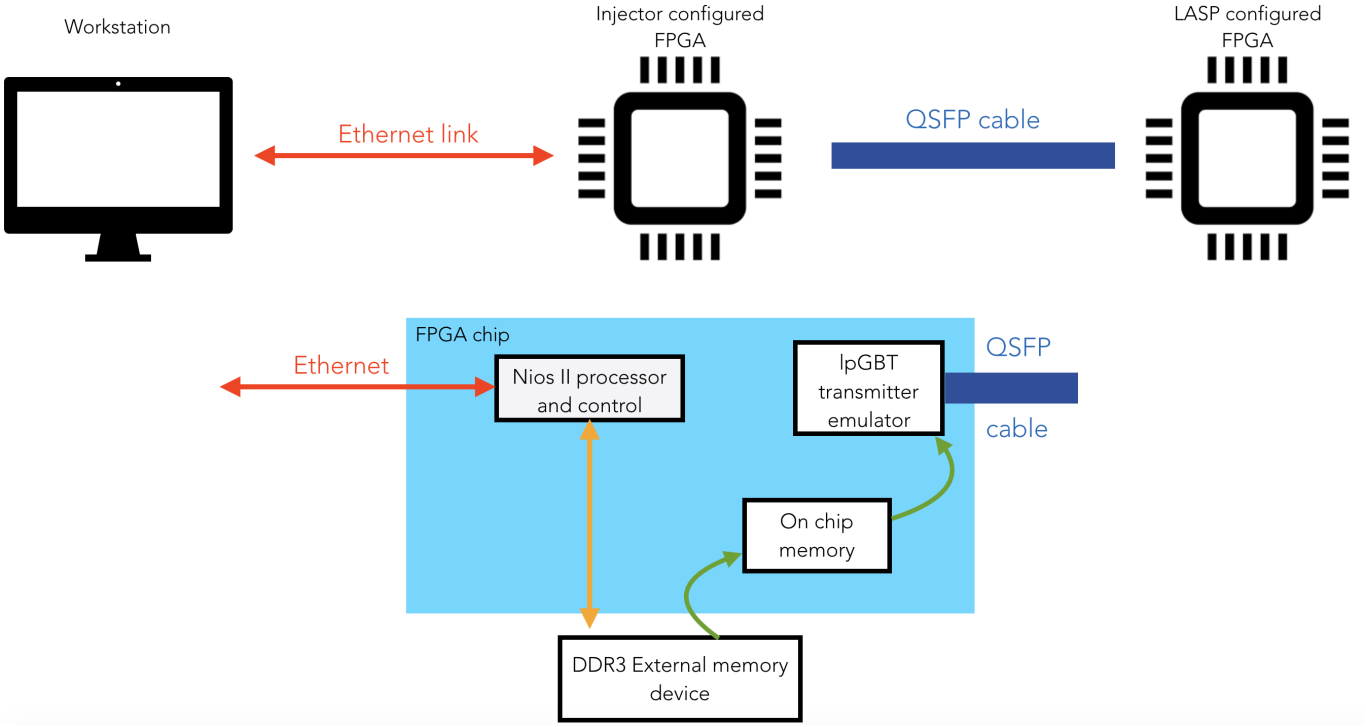


Figure 6.1: Simple schematic of the Injector project and how the framework is intended to be used. Arrows indicate the data-path from workstation to LASP

a slow communications protocol using the low-speed, and short transmission length RS-232 standard. For this reason, the Ethernet protocol was chosen to be the channel through which AREUS data can be transferred to the FPGA. A JTAG UART communication link between the workstation and the injector-implemented FPGA is also used for other control and status communications.

To simplify the process of coordinating the Ethernet connection, the FPGA is configured with a **NIOS II** processor (see Chapter 5.3.3). The processor is responsible for setting up an Ethernet **Socket** server and then receives AREUS data.

Because the Intel Stratix 10 GX FPGA board has an internal memory capacity of 28 MB [36], the amount of AREUS data that can be stored here is quite small. In order to have an effective data injector, one needs larger amounts of AREUS data which will result in longer periods of injection to the LASP. Fortunately, the Stratix 10 FPGA has an off-chip **DDR3/DDR4** memory device. This can be used to store larger quantities of AREUS data.

The Injector project uses a 2GB external DDR3 memory device to store AREUS data. To coordinate this transfer, the NIOS II processor is once again employed. The

processor will hence both receive the data and store it onto the DDR3 device.

Since the Injection and the Storage sub-modules use the NIOS II processor, both hardware and software code must be written for them. The Hardware description specifies the instantiation of the processor and the other hardware protocols needed to fulfill the tasks. The software component is a standard `C` program which is executed on the NIOS II processor.

The Injection and the Storage sub-modules are thus embedded systems. To aid the design of such embedded systems, the **Quartus Prime** tool - **Platform Designer** (see Chapter 5.3.4) has been used. Figure 6.2 shows the grand scheme for the Injector project, highlighting the Injection and the Storage sub-modules being implemented as Platform Designer sub-systems.

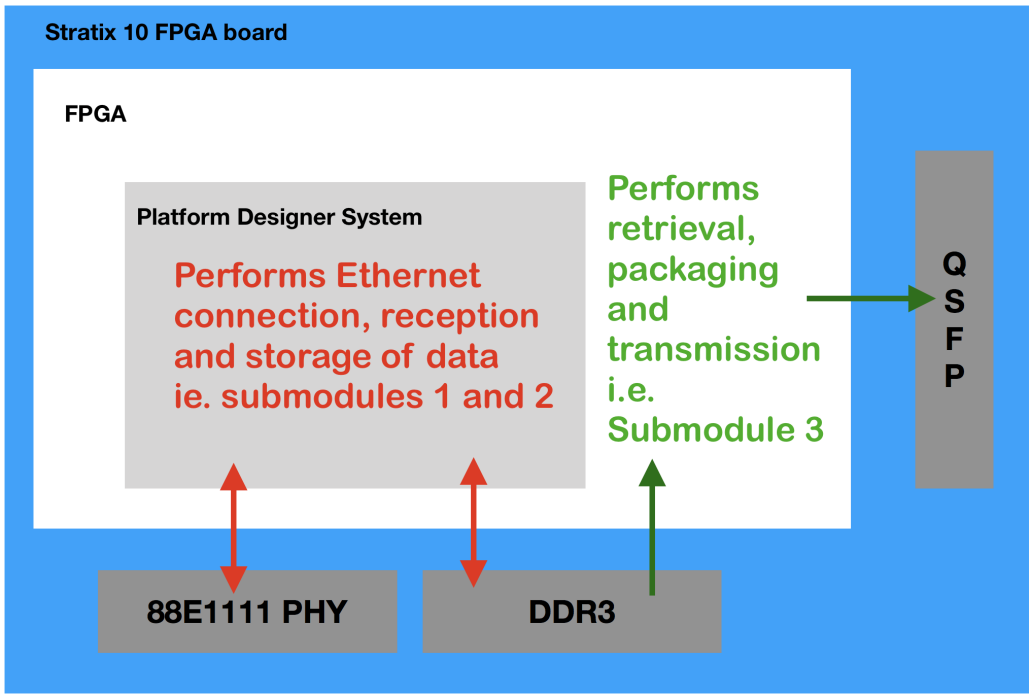


Figure 6.2: An extended schematic of how different sub-modules in the Injector Project are implemented. Sub-modules within the Platform Designer System have both a hardware and a software component. Sub-modules outside of the Platform Designer System have a hardware component alone. 88E1111 PHY refers to the Physical chip that provides Ethernet support.

Once the DDR3 is filled up with data, it is then read and stored into on-chip memory buffers. This marks the beginning of the data retrieval and transfer sub-module. The small size buffers are refreshed continuously and data is then sent to the **lpGBT** transmitter emulator.

The lpGBT transmitter emulator is a module designed by the lpGBT group at CERN. The module consists of transceiver capabilities and error correction and data-frame aligners for the payloads. The module then transfers 230-bits of user data at 10.24 Gbps (thus "emulating" the lpGBT chip) to the LASP-configured FPGA with the use of **Quad small form-factor pluggable (QSFP)** cables. The data retrieval and transfer sub-module is implemented through hardware only as shown in Figure 6.2.

6.2 Data Injection

The first sub-module in the Injector Design architecture is the transfer of data to the injector-configured FPGA. The **Ethernet** communications protocol is used for this transfer because it is fast and flexible. A workstation connected to a **Local Area Network (LAN)** running the Ethernet protocol is able to transmit and receive data to and from an FPGA connected to the same network.

6.2.1 The Ethernet Protocol in Stratix 10 FPGAs

Ethernet is a relatively inexpensive and reasonably fast LAN technology that was first used in the 1970s. Currently, Fast Ethernet speeds have extended the performance of previous generations of the technology: now reaching speeds of 100Gbps! These products are not only expensive but require additional infrastructure that is not available to every-day research facilities yet.

With respect to the **Open Systems Interconnect (OSI)** model, Ethernet operates across the first two layers: the Physical Layer and the (lower half) of the Data Link layer as shown in Figure 6.3.

The Ethernet implementation is done in the **Media Acess Control (MAC)** sub-layer (which lies in the lower half of the Data Link layer in the OSI model) and the **PHY (Physical)** layer. The MAC performs data encapsulation including frame assembly before transmission and frame parsing upon reception of a frame. The MAC is also responsible for the recovery of frame transmission in case of collisions. The upper half of the Data Link layer is called the **Logical Link Control (LLC)** sub-layer. The LLC, which is implemented in software, handles communication between the upper layers and the MAC.

The PHY layer consists of the physical transceivers and provides a link between the

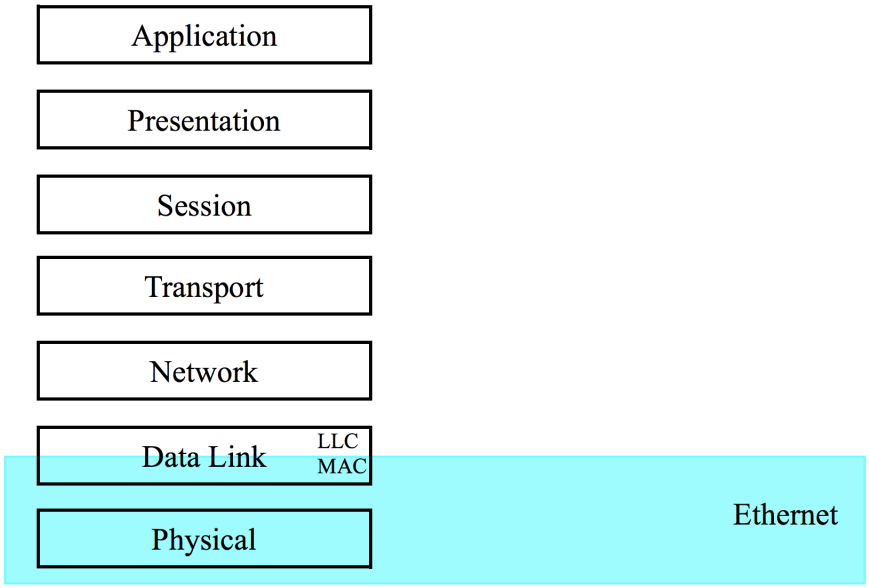


Figure 6.3: Block diagram showing the Ethernet Standard within the OSI model. Diagram taken from [41].

MAC layer and the twisted pair cable. The PHY layer is responsible for establishing a functioning link and choosing common transmission parameters.

The Intel Stratix 10 FPGA provides Ethernet support by using an external Marvell 88E1111 PHY chip, which is a physical layer device integrated with a 10/100/1000 Mbps Ethernet transceiver. The MAC layer, used to communicate with this chip, is implemented by an Intel IP core called **Triple-Speed Ethernet (TSE)**.

6.2.2 Ethernet Hardware Development

Hardware development of a complicated embedded system is difficult and time consuming to construct from scratch. A common method employed for solving known problems is to use **IP** cores that are offered by designing parties such as Intel. The IP cores are either delivered in high-level **HDL** codes or can be within a detailed transistor-level layout [31].

Platform Designer (see Chapter 5.3.4) is used to define the embedded system, connect the sophisticated bus structure, and generate HDL codes of the system. Figure 6.4 shows the design configuration for the Ethernet sub-system. Below is a description of each component used in the sub-system.

Imported clock signals `sysClk` (C1) and `eth_refClk` (C2) run at 80 MHz and

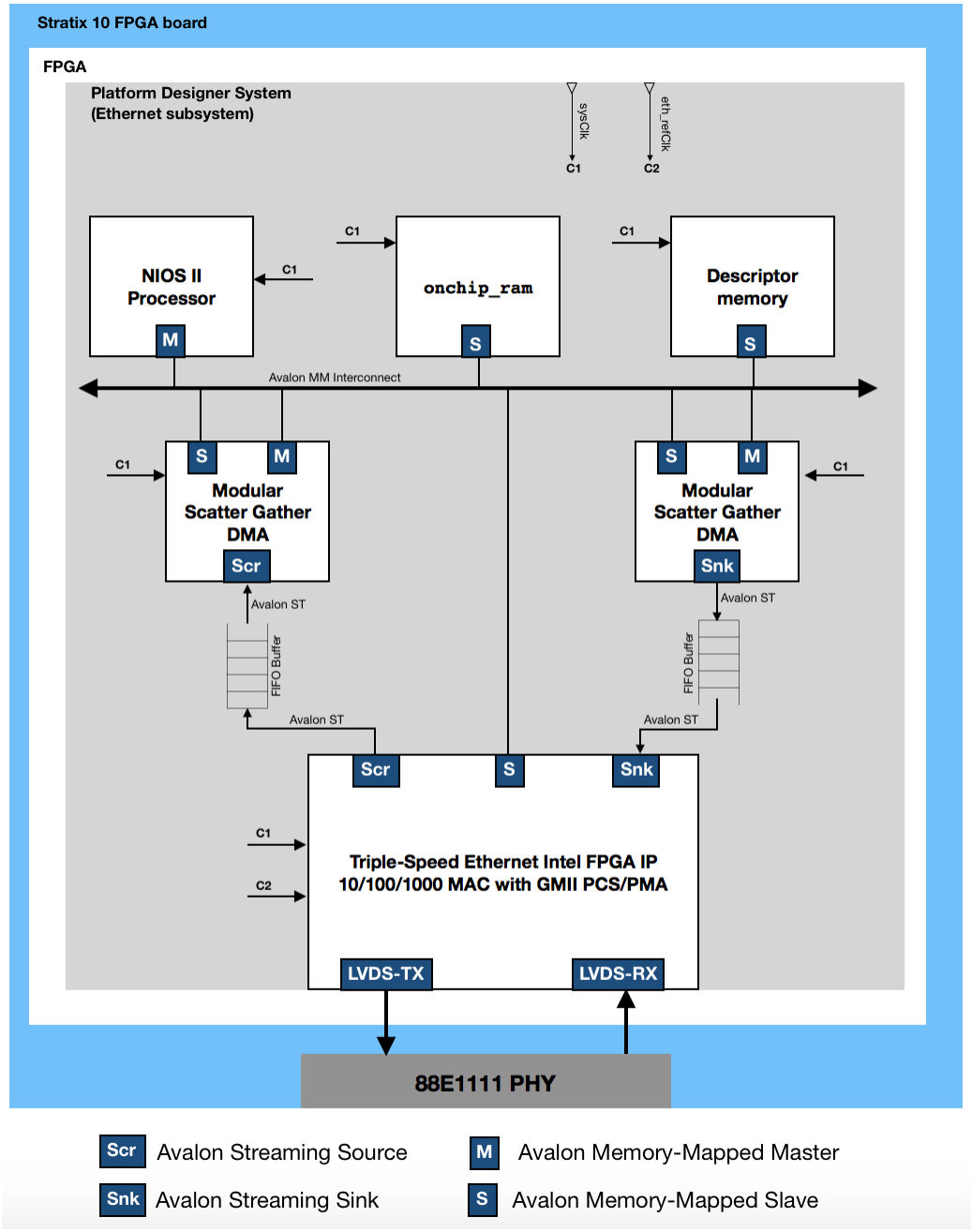


Figure 6.4: Schematic of the Ethernet subsystem. Data flowing to the Ethernet PHY chip is received and processed in Triple Speed Ethernet IP which provides the MAC layer. The configuration is made such that the data received is stored in an on-chip RAM. The NIOS II processor controls and monitors the running of the whole subsystem. LVDS refers to Low-voltage Differential Signaling I/O transceivers.

125 MHz, respectively. C1 clocks many of the subsystem components while C2 is used solely to clock the interface that connects the Ethernet PHY to the MAC.

A `NIOS II processor` is also instantiated and optimized for faster performance. Since it is an Avalon Memory Mapped Master, it has the ability to control an Avalon Memory Mapped Slave. The NIOS II is connected to the slave ports of (and can therefore control) all the other components in this subsystem. The processor's instruction code needed for boot-up is stored in the the `onchip_ram`.

The `onchip_ram` is a 2 MB on-chip memory storage. It is allocated to control the flow of data from the Ethernet and to hold the NIOS programs. The on-chip memory is a crucial component as it receives data from the Scatter-Gather **DMAs** which facilitate the transfer of data from the Ethernet. The memory type is RAM (writable) and hence the processor can make changes to the contents, if needed.

The `Triple Speed Ethernet` **(TSE)** IP provides the HDL instantiation of the MAC and upper sub-layers of the PHY. The TSE supports 10/100/1000Mbps Ethernet operations. The interface between the MAC and the upper sub-layers of the PHY is done via **GMII (Gigabit Media-Independent Interface)**.

Data received by the TSE from the Ethernet is in a streaming interface. To store the data into the memory (on-chip or off-chip), this must be converted to a memory-mapped interface. The Receiver `Modular Scatter-Gather DMA` **(mSGDMA)** does this. The transceiver `Modular Scatter-Gather DMA` transfers data from a memory-mapped location to a streaming interface. The movement of data by mSGDMAs is done using a buffer called a **FIFO (First In, First Out)**.

Modular DMAs are capable of performing data movement operations with preloaded instructions known as descriptors. Instructions on the source and destination address for each mSGDMA is stored in the `descriptor_memory` component.

Data that is received from the TSE is passed onto the receiving mSGDMA. This mSGDMA transports the data from a streaming form to a memory-mapped form and stores it in the onchip_ram. The reverse is done for the transmitting process. The whole path is initialized, monitored, and under the control of the NIOS II processor.

6.2.3 Ethernet Software Development

The above hardware description sets the Ethernet interface up until (and including) the MAC layer. The upper half of the Data Link in the OSI model i.e. the LLC is implemented in software. All the other upper layers from the LLC are also implemented in software.

In this project, the **Internet Protocol (IP)** will be used for the implementation

of the network layer. The protocol defines the format of packets and provides an addressing system for routing datagrams from a source host interface to a destination host interface. The transport layer will be implemented by the **Transmission Control Protocol (TCP)**. The sophisticated handshaking mechanisms and error-checks present in TCP ensures a reliable stream of data. Together these two layers form a TCP/IP stack.

The upper layers of the **OSI** model can be amalgamated to form one "Application" layer. The Application essentially has to be able to setup a so called **socket** - an internal end-point in the TCP/IP connection, after which it can then send and receive data on the network. Sockets can either be servers or clients. The socket server is responsible for establishing a valid IP address bound to a specific port number. The server then waits, listening for a client to make a connection request. The Injector based FPGA is designed to be a socket server.

Figure 6.5 describes the software architecture model that is used to run on the NIOS II processor to implement the Ethernet connection. Each layer is explained below:

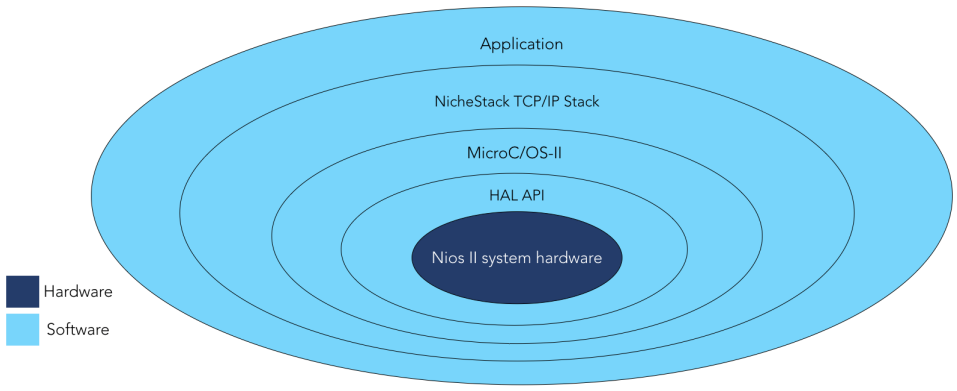


Figure 6.5: An onion diagram showing the architectural layers of Ethernet Implementation on the FPGA. Each layer abstracts the data for the next outer layer. Services from all the inner layers are harvested by the Application layer to be used in the implementation of the main software task. Diagram adapted from [42].

HAL API

The **Hardware Abstraction Layer (HAL) Application programming interface (API)** provides a standardized interface as well as device drivers and a standard C library. Device drivers include routines that know the physical details to function with respective hardware such as with the TSE.

MicroC/OS-II

MicroC/OS-II is a real-time operating system multitasking kernel. It provides multitasking and communication abilities to the Application layer and the TCP/IP stack.

NicheStack TCP/IP Stack

For the NIOS II processor, the NicheStack TCP/IP Stack is a small footprint implementation of the TCP/IP suite. The focus of the NicheStack TCP/IP Stack implementation is to reduce resource usage while providing a full-featured TCP/IP stack [43]. The NicheStack TCP/IP Stack layer provides networking services to the application layer through socket API.

Application

The outermost application layer has three main tasks: Setting up the Socket server (using the services from the inner layers), receiving data from the TCP/IP connection, and storing data into the **DDR3** (which is part of sub-module 2). Details on the first two tasks are given below. A full description of the whole application software is deferred to Section 6.4.2, after discussion of the Data Storage sub-module.

6.2.4 Ethernet Specific Application Software

For the Injector project, the socket server is created on the Stratix 10 GX FPGA. The client request comes from a workstation connected to the same Ethernet network as the FPGA. The Ethernet Socket Server running on the NIOS II processor uses the industry standard sockets interface to TCP/IP. It uses the **DHCP** protocol to request a valid IP from the Gateway. During the device initialization process, the MAC address for the FPGA board is generated. Autonegotiation is then initiated where both connected devices, the Ethernet PHY (Marvel 88E1111), and the gateway device broadcast their transmission parameters, speed, and duplex mode. Once the link is established an IP address is assigned to the Ethernet device along with the port number. The TCP/IP stack is now fully operational and the MicroC/OS-II can now handle application-level tasks such as receiving the data. The only information needed for a client to connect to the Socket Server is the IP address and the port number. Both of these are displayed by the Socket Server. Further details about how the client connects to the socket is deferred to Section 6.4.2

The `SSSSimpleSocketServerTask()` function, which is called in `simple_socket_server.c`,

sets up and manages the socket server by doing the following:

- Using socket API to create a TCP/IP socket endpoint

```
fd_listen = socket(AF_INET, SOCK_STREAM, 0)
```

  where `AF_INET` setups the IPv4 Internet protocols and `SOCK_STREAM` provides a sequenced, reliable, two-way, connection-based byte stream. The argument zero describes that only one protocol exists in the creation of the socket.

- Using socket API to bind a name to a socket

```
bind(fd_listen,(struct sockaddr *)&addr,sizeof(addr))
```

  where `addr` is the address that is assigned to the newly created socket `fd_listen`.

- Using socket API to listen for TCP/IP connection requests from a client

```
(listen(fd_listen,1)
```

  where the socket `fd_listen` is being listened to with a maximum backlog of 1 connection.

- Calling `sss_handle_accept()` during an incoming TCP/IP connection request. A connection will only be accepted if no other client has established a connection with the server. If a connection is accepted, `sss_send_menu()` is called which sends a menu of available options to the client.

- Calling `sss_handle_receive()` to serve the TCP/IP connection. This function calls `sss_exec_command()` which is where the application code lies for processing and storing the data transmitted via the socket.

6.3 Data Storage

Data has now been transferred to the **NIOS II** processor via the Ethernet link. The data that is received by the processor is temporarily mapped (by the **mSGDMA**) to the 2MB On-chip memory `onchip_ram_m9`. The data is then transferred to the external memory device by the processor. For the Injector project, a 2GB Micron **DDR3** external memory device is used.

6.3.1 DDR External Memory devices

As mentioned in Section 5.3.2, the Stratix 10 GX FPGA has a Hard Memory controller and a HiLo connector that can support DDR3 and DDR4 memory daughtercards.

DDR memory devices, more formally known as DDR SDRAMs, are **double data rate (DDR) synchronous dynamic random-access memory (SDRAM)** data devices that are used extensively as integrated circuits since their invention in 1997. As it is synchronous, control signals are detected at the rising edge of every clock. SDRAM memory is divided into several independent sections called banks, which allows simultaneous memory access to multiple banks thus speeding up performance.

DDR3 SDRAM is the third generation of SDRAM. DDR3 SDRAM is internally configured as an eight-bank DRAM and uses an $8n$ prefetch architecture to achieve high-speed operation. This architecture is combined with an interface that transfers two data words per clock cycle at the **I/O** pins. A single read or write operation for DDR3 SDRAM consists of a single $8n$-bit wide, four-clock-cycle data transfer at the internal DRAM core and eight corresponding n-bit wide, one-half clock cycle data transfers at the I/O pins.

Read and write operations to the DDR3 SDRAM are burst oriented. This means that for a given address, reads/writes are done for that address along with multiple consecutive addresses. Typical operation of DDR3 begins with the registration of an active command, which is followed by a read or write command. The address bits registered coincident with the active command select the bank and row to be activated. The address bits registered coincident with the read or write command select the starting column location for the burst operation [44].

6.3.2 DDR3 Interfacing with NIOS II - Hardware Subsystem

Intel has developed a low-latency **External Memory interface (EMIF)** IP core which allows interfacing with the memory devices. The IP core provides two components [45]:

- A physical layer interface (PHY) which builds the data path and manages timing transfers between the FPGA and the memory device.
- A memory controller which implements all the memory commands and protocol level requirements.

Figure 6.6 shows the design configuration of the DDR3-interfacing subsystem.

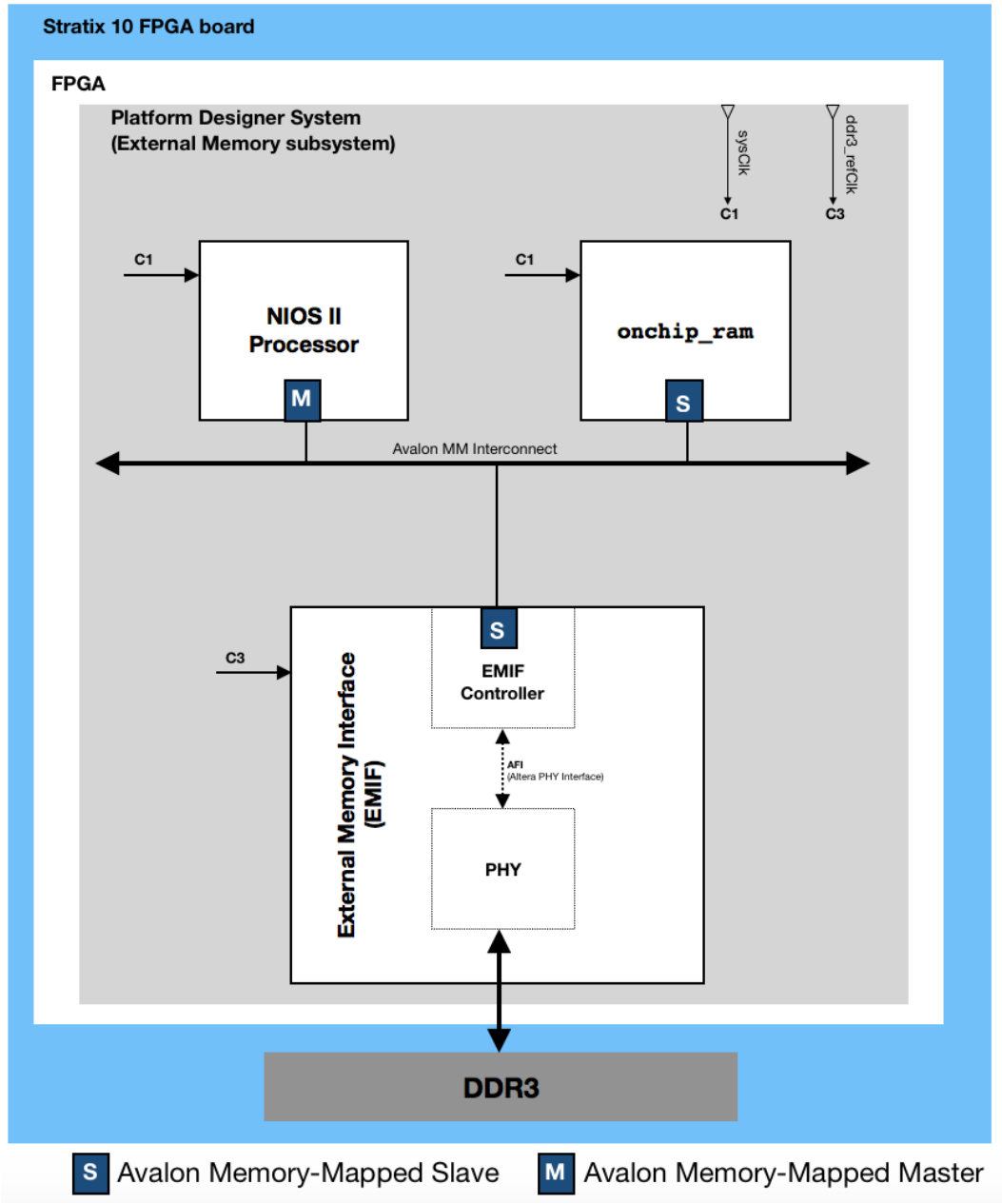


Figure 6.6: Schematic of the External-memory subsystem. C1, the NIOS II Processor and the Onchip RAM are the same components from Figure 6.4. The PHY layer of the EMIF and the memory controller communicate via the Avalon PHY Interface.

The same 80 MHz `sysClk` (C1) is imported in this subsystem as well. A new 133.33 MHz `ddr3_refClk` is also imported into this subsystem to clock the EMIF. Using an internal PLL, the DDR3 I/O bus is clocked 8 times faster than the EMIF i.e. 1066 MHz.

The `NIOS II Processor` and the `onchip_ram` are the same components described in Figure 6.4 and are shared between the two subsystems.

The `EMIF` has a memory controller that acts as an Avalon Memory-Mapped Slave. This means that an Avalon Master, such as the NIOS II processor, would be able to send and receive read/write commands. Thus, data that is received by the `rx sGDMA` from the Ethernet is buffered in the `onchip_ram` and is then transferred to the EMIF by the NIOS II processor. The reason the on-chip memory buffer is needed is because the logical addressing is much more simpler and intuitive in this memory device than the physical addressing in the DDR3.

6.3.3 DDR3 Interfacing with NIOS II - Software

The software component for the External Memory interface is less involved than the Ethernet implementation. With regards to the software's architectural structure of Figure 6.5, the DDR3 interfacing falls under the "Application" layer.

The **HAL API** layer includes a header file called `system.h` which encapsulates all the components of the hardware system including the component's respective memory addresses. For the NIOS to DDR3 data transfer, important variables are `EMIF_S10_0_BASE` and `ONCHIP_RAM_BASE` which holds the base address of the DDR3 chip and the the On-chip RAM, respectively.

The application code can be found in `simple_socket_server.c` (Appendix C), where preliminary tasks that need to be achieved before execution of this code include creation of the Ethernet and establishment of a handshake between client and server. The client then sends the AREUS data which is received by the processor.

Code that runs on the NIOS II processor and manages the receipt and storage of the data to the EMIF is shown below:

```
1  while(received_bytes < expected_size && received_status > 0)
2  {
3    received_status = recv(conn->fd, pBuffer,
4                           expected_size - received_status,0);
5    memcpy(EMIF_S10_0_BASE+received_bytes,pBuffer,received_status);
6    pBuffer += received_status;
7    received_bytes += received_status;
8  }
9  alt_dcache_flush_all();
```

`received_bytes` and `expected_size` are `int`s that track the number of bytes the

connection has received thus far and the expected total number of bytes the connection has to receive, respectively. `received_bytes` is initialized to zero. `received_status` is a status `int` that registers the length of the message received in bytes. It is initialized to a value of 1.

The while loop continues until all the data has been received from the client. The `recv` function call from the Socket API receives messages from the socket `conn->fd` and stores the received message in the address pointed to by `pBuffer` which is of length `expected_size - received_status`. `pBuffer` initially points to `ONCHIP_RAM_BASE`. The `received_status` variable is now updated with the number of bytes received from that particular function call. It is important to note that TCP/IP send and receive data in slightly irregular packets. Thus, the server needs to keep on listening for newer packets and update accordingly.

The `memcpy` function call copies `received_status` number of bytes from the memory address pointed by `pBuffer` to the memory address pointed by `EMIF_S10_0_BASE + received_bytes`. The pointer `pBuffer` and the cumulative `received_bytes` variable increments by `received_status` number of bytes. The whole process is repeated until the while loop conditions are not satisfied.

The `alt_dcache_flush_all()` function flushes i.e. writes back dirty data and then invalidates the entire contents of the data cache. This ensures that all the data is copied into the DDR3.

6.4 Full Hardware and Software Interface

6.4.1 Peripheral hardware components

For the full functionality of a NIOS II system, other peripheral hardware components are needed that have been excluded in the prior discussions as they did not support the implementation of the Ethernet setup or the DDR3 interfacing. Nonetheless, these components are crucial to creating a complete NIOS II system and their design configuration is as illustrated in Figure 6.7.

The same `sysClk` (C1) is once again imported to this subsystem to clock all of the components residing here. C1 oscillates at 80 MHz.

The `NIOS II Processor` and the `onchip_ram` are also the same components that are described in Figures 6.4 and 6.6. These two components are shared between all 3 subsystems. The NIOS II controls all the other components in this subsystem, each

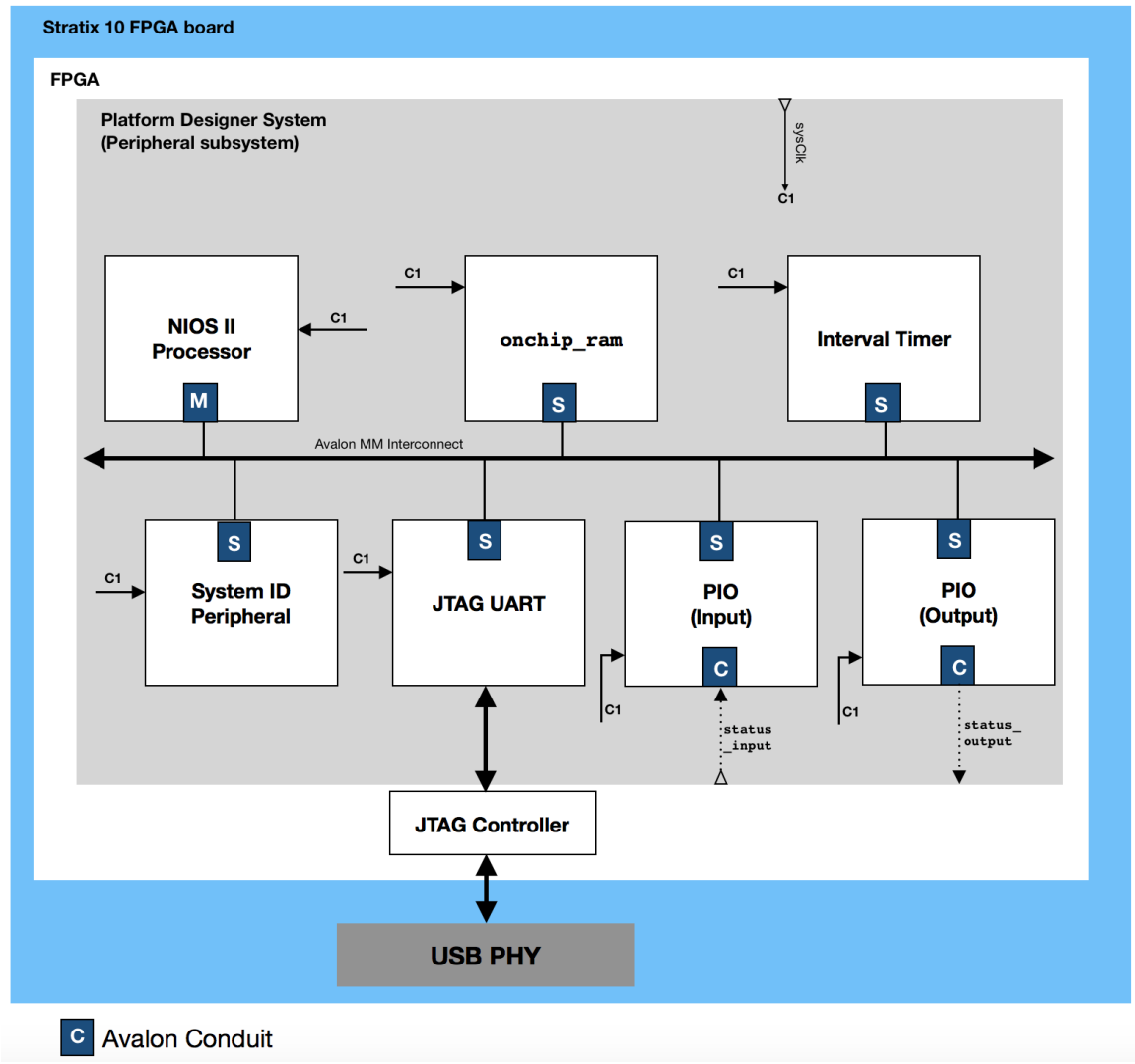


Figure 6.7: Schematic of the Peripheral subsystem. C1, the NIOS II Processor and the Onchip RAM are the same components from Figures 6.4 and 6.6. Signals `status_input` and `status_output` are imported/exported to the other parts of the FPGA code.

offering unique characteristics that are useful for the overall running of the embedded design.

The `Interval Timer` is used to support timing needs, such as measuring the interval between events and also for generating periodic pulses. It is used by the TSE as well as the MicroC/OS-II to monitor timing tasks.

The `System ID Peripheral` is a special-purpose peripheral used to maintain the consistency between the hardware configuration and software. It is essentially a 32-

bit signature that is added to a particular memory address in the `onchip_ram_m9`. When the software files are being created for a particular hardware, this signature is passed into the software image that will run on the NIOS II. While downloading the software image, NIOS II checks whether the signature of the system ID module in the FPGA device matches the signature embedded within the software image and thus ensures the consistency between the hardware and software.

The `JTAG UART` core is similar to a serial port that is used to communicate the serial character stream between the PC and the board. Moreover, it is used to download the software image to the NIOS II processor. The **JTAG UART** also has Interrupt Request capabilities, which can be used to debug NIOS II programs running on the processor.

The `PIO` **(Parallel Input/output)** core provides an Avalon Conduit Interface which can be imported/exported to either a general-purpose **I/O** port or to other hardware code. In this context, the imported and exported signals will indicate who is using the DDR3.

The `status_output` is a 1-bit wide output status register that will be written by the NIOS II processor. When the NIOS II processor is busy i.e. receiving and transferring data from the Ethernet to the DDR3, the output will be de-asserted (low). When the NIOS II processor has finished transferring the data, this wire will be pulsed i.e. asserted high and then quickly deasserted again.

This status register will be checked by other parts of the Injector to ensure that the DDR3 is not being used when busy. This is especially important because the reading of the DDR3 in sub-module 3 needs to be done only when the NIOS II processor has finished using the resource.

The `status_input` is a 1-bit wide input status register. The NIOS II processor cannot write to this register but can only read it. This register is asserted (high) when the DDR3 resource is used by another component other than the NIOS II processor. This then instructs the NIOS II processor to halt any commands accessing the DDR3 to avoid resource conflict. The opposite is true when this status register is deasserted.

6.4.2 Full Software algorithm design

So far, we have elaborated on how the software running on the FPGA sets up a **TCP/IP Socket** server, receives data and stores that data into a DDR3. This thesis also provides a compatible program, `bulk_client_send`, executed on the workstation,

that requests a connection with the FPGA and then sends data. Figure 6.8 shows the algorithm design of the program that runs on both the FPGA and the workstation.

In the NIOS II program, the MicroC/OS-II is started up and is given a task to initialize the stack. This task is assigned a high priority and is the first one the processor attends to. The functions `alt_iniche_init()` and `netmain()` are then called which initializes the stack and manages the MAC and IP addresses for the network interface. Once the initialization process is complete, the global variable `iniche_set_ready` is assigned to be `TRUE`.

Once this happens, the MicroC/OS-II is assigned a new task: `SSSSimpleSocketServerTask`. This task creates and manages the socket server connection, and calls relevant subroutines to manage the socket connection. Once such task is `sss_send_menu()` which, after successful connection, sends a welcome menu to the client. The welcome menu has the following options:

```
1 ==================================
2 LASP Injector Socket Menu
3 ==================================
4 ACQUIRE      - Obtain data from Workstation
5 TRANSFER     - Transfer data to H-tile
6 MENU         - Display this menu
7 QUIT         - Quit
8 ==================================
```

The software running on the workstation is programmed to receive this menu and select the option Acquire. Once this command is received by the NIOS II processor, the program stalls until it receives an `int` of how many **ADC** values the client wishes to send. The program in the workstation is set to send a value that corresponds to 1.8 GB of ADC data. Once this `int` is received, the workstation then sends the ADC data itself. This is received by the NIOS II and transferred to the DDR3. This continues until the expected number of ADC data is received by the NIOS II.

The workstation then issues the command Transfer. This transfer command causes the `status_output` **PIO** peripheral to generate a low-high pulse. This pulse is detected by the FPGA logic and is a signal that the NIOS II processor is not using the DDR3 memory resource anymore. The workstation then issues the command Quit, which closes the socket connection.

The NIOS II program now stalls in the `SSSSimpleSocketServerTask`, awaiting an incoming connection from a client. The program in the workstation returns and closes.

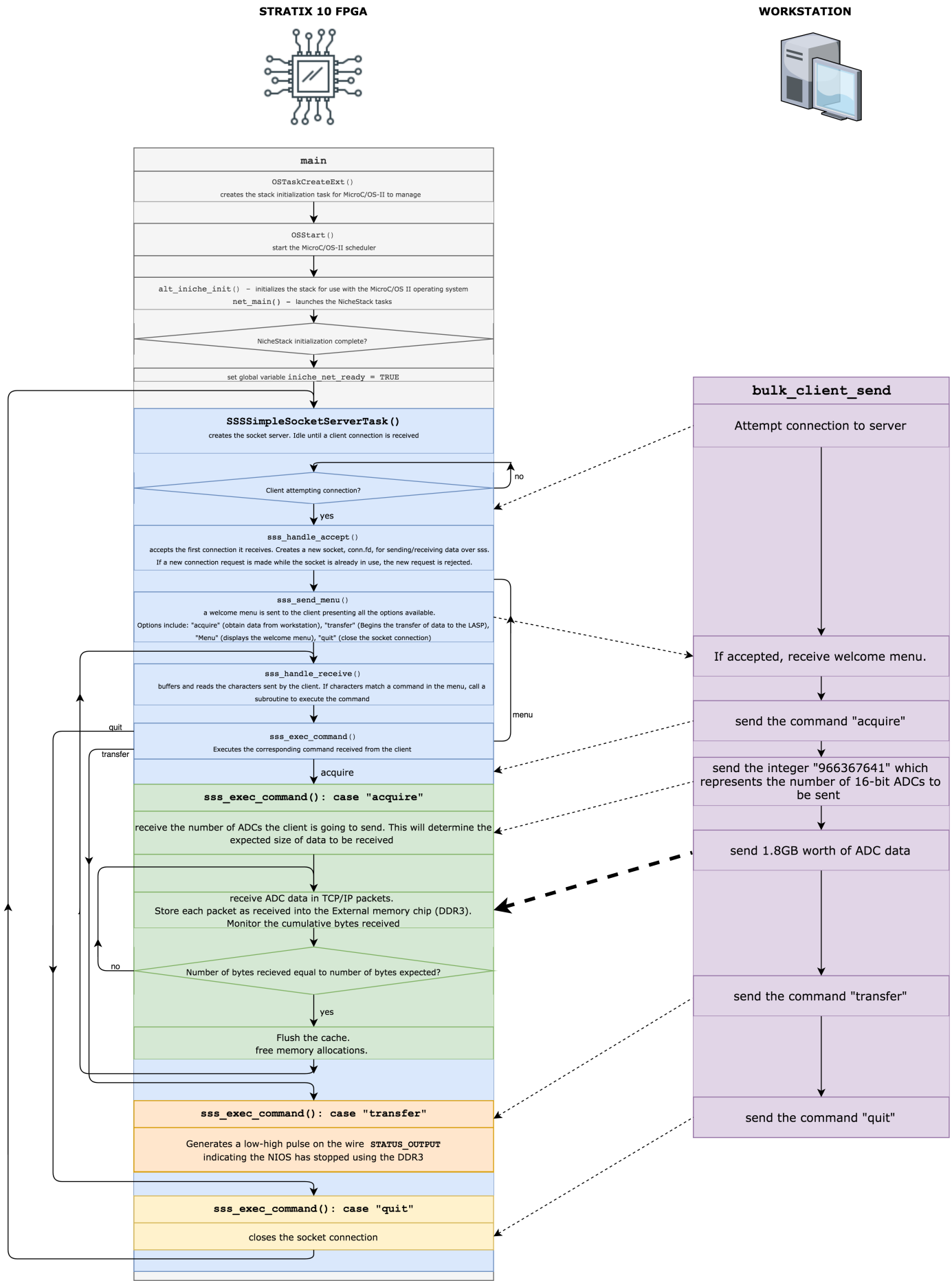


Figure 6.8: Complete Algorithm design for software in both the FPGA and the workstation. The dotted lines represent TCP/IP packets.

6.5 Data Retrieval and Transfer

Data has now been transferred from the workstation to the FPGA and is then stored in an off-chip DDR3 memory device. The next part of the Injector project is to extract this data and send it to the LASP in a payload size of 192 (user controlled bits) and at a payload rate of 40 MHz.

Because the NIOS II processor is clocked at 80 MHz, a system where data is retrieved using the processor will be too slow. The NIOS II system is a favourable way to implement the Ethernet connection, but with regards to data retrieval, using the hardware to do the job is much more efficient. For this reason, this stage of the Injector project is hard-coded onto the fabric of the FPGA and does not rely on a processor to control it.

6.5.1 Data Retrieval

The contents on the **DDR3 SDRAM** are accessible via the **Avalon Interface**, in a similar manner to the way the NIOS II processor interfaces with the DDR3 in the data storage stage in Section 6.3.2. To interconnect via the Avalon Interface, a new IP core which can connect to the DDR3 has to be used.

Master Template IP

Intel does not provide standard IP solutions for interfacing with an External Memory Device, and so for this task, a non-standard IP had to be adapted. Since the DDR3 has an Avalon Slave component, the IP that is to be adapted must be an Avalon Master i.e. can offer read/write commands to the slave. The IP that is tasked with doing this in the Injector project is named “Master Template IP”.

Because the data has to be post-packaged into an lpGBT frame, the IP must be able to extract the data contents of the DDR3 so that it can be further used by the user logic code. The user logic code must also be able to dictate when to start reading the DDR3 since it is important that the read does not happen when the NIOS II is writing to the DDR3. User logic code also needs information on when the IP has completed the read.

The Master Template IP is implemented using an internal **FIFO** buffer, which pushes results of the DDR3 reads into the FIFO and pops them out when user-logic has requested for them. The full Master Template IP block diagram is shown in

Figure 6.9 with the associated signals explained in Table 6.1.

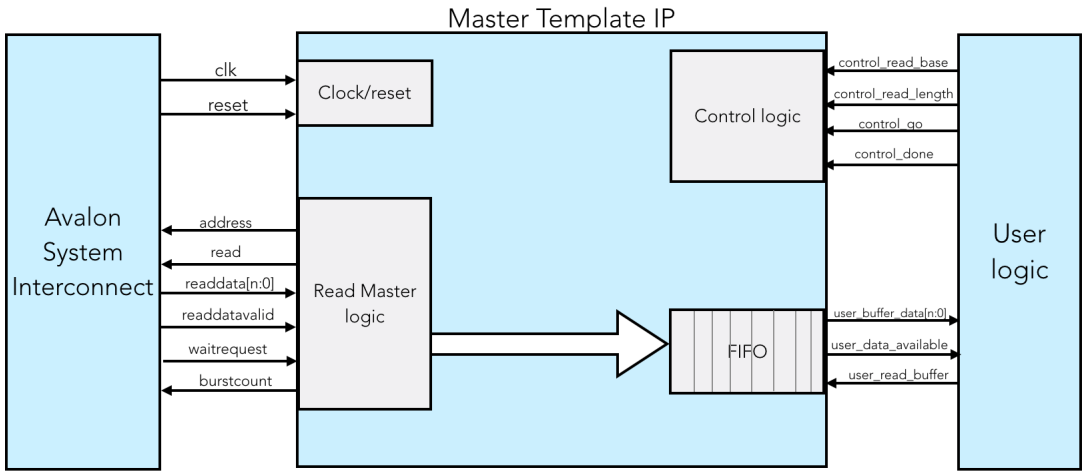


Figure 6.9: Block diagram showing the signal interfaces of the Master Template IP. Signals interfacing with user logic are explained in Table 6.1, while signals interfacing with the Avalon Interconnect are those described in Chapter 5.3.4. Diagram adapted from [46].

Signal Name	**Direction**	**Width (bits)**	**Usage**
`control_read_base`	Input	32	The first word address where the Master IP will start transferring data from
`control_read_length`	Input	32	Number of bytes to transfer
`control_go`	Input	1	Instructs the master to begin transferring
`control_done`	Output	1	Asserted when the master has transferred the last word of data
`user_data_buffer`	Output	n where n is a power of 2	Contains the buffered data from the DDR3
`user_data_available`	Output	1	Asserted when the user buffer contains valid data that has been read from the DDR3
`user_read_buffer`	Input	1	Acts as a read acknowledge and pops the first buffered data from the FIFO.

Table 6.1: Description of signals interfacing the Master template IP and the User logic. Table adapted from [46].

The Master Template IP is instructed by user logic at which address to start the read (`control_read_base`) and how many bytes to read (`control_read_length`).

When `control_go` is triggered to start, the Master template IP asserts a `read` command to the Avalon interface connecting to the DDR3 and issues the `address` for which it should read as well as the `burstcount`. When a valid read is made, indicated by assertion of `readdatavalid`, the contents in `readdata` are pushed inside a FIFO. This continues until the FIFO is full, at which point the `read` signal is de-asserted.

When the FIFO has data, `user_data_available` is asserted. When the data contents `user_data_buffer` has been read, `user_read_buffer` is also asserted, which causes the FIFO to remove/pop its oldest data entry i.e. the data that was first read. At this point, the FIFO is no longer full and can accept more pushes from the Read Master Logic.

The Read Master Logic increments the `address` for each `read` command, taking into account the `burstcount`, to make sure the entire length is read. Once the length is read, the Read Master logic stops issuing commands to the Avalon interface and asserts `control_done`, which signals to the user logic that the complete read is finished.

One limitation of using this method to read the DDR3 is the fact that reads are limited in width size to be a power of 2. The required data size for the lpGBT payload is 192 bits, which suggests either using $n = 128$ or $n = 256$, where n is the bit width of the `user_data_output`. Another consideration is the throughput of the read. Since the DDR3 is an external memory device, data transfer takes a longer route to the FPGA chip and is thus slower. Tests have shown that with $n = 64$ and the Master Template IP core clocked at 80 MHz, the throughput obtained is $\sim$200 MBytes per second. The LASP requirement is 192 bits every 25 ns i.e. a throughput of 960 MBytes per second. The above implementation would therefore need to be modified to meet the design criteria.

The Injector project works around this limitation by using 3 independent instances of the Master Template IP. Each IP then performs a 512-bit read resulting in a total read of 1536 bits. The master logic is modified so that the address reads for each IP do not overlap with each other. The 3 IPs perform a sequential read. This design obtains an impressive throughput of $\sim$4 GBytes per second. Interfacing of the three Master Template IPs with the **EMIF** is described in the Data-retrieval subsystem shown in Figure 6.10.

`sysClk` (C1) and the `EMIF` are the same components shown in Figure 6.6. All of the `Master Template` cores are clocked at 80 MHz. The `Master Template` has the ability to issue read commands to the EMIF which in turn interfaces with the DDR3

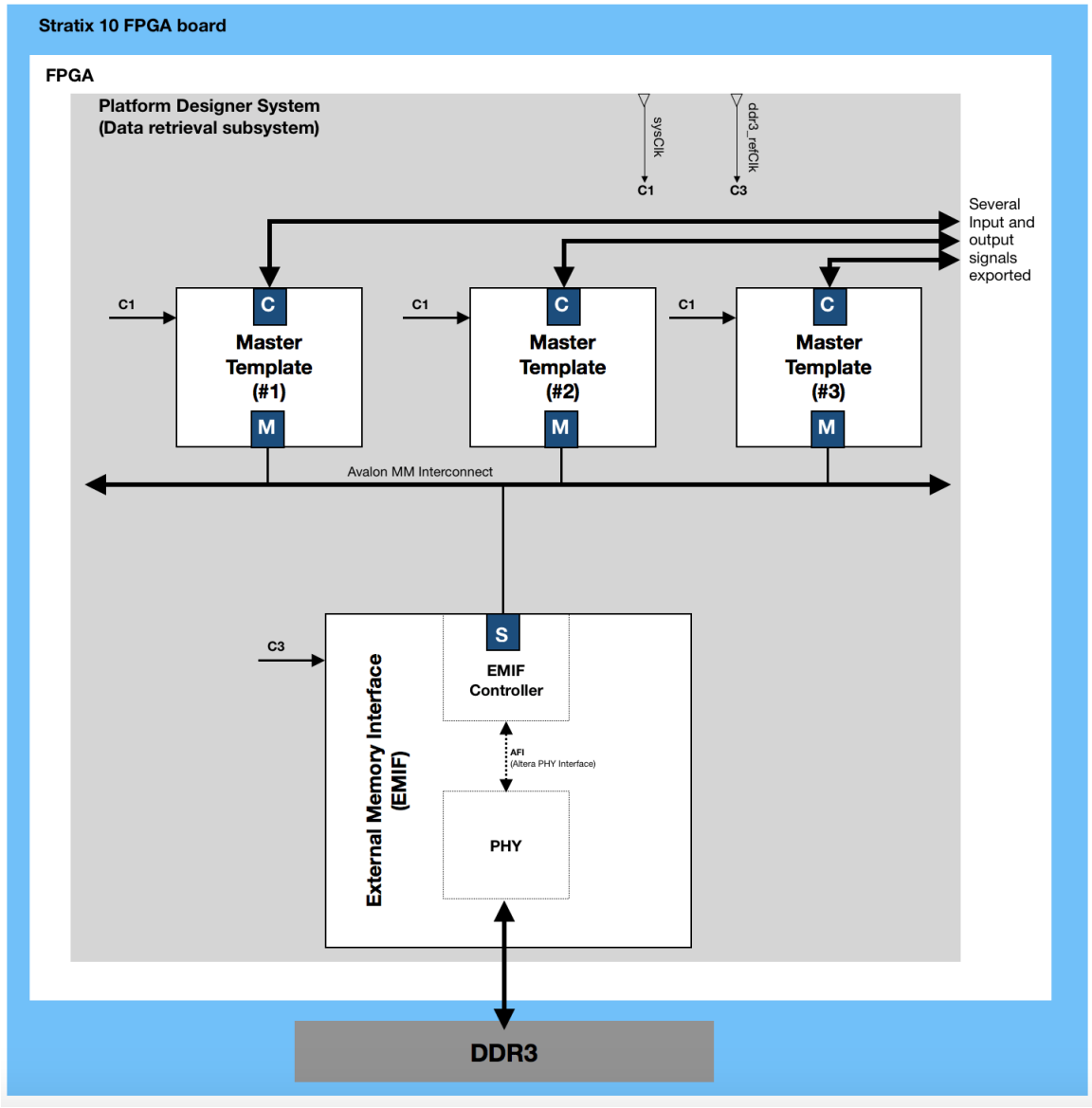


Figure 6.10: Schematic of the Data Retrieval subsystem. C1 is the common 80 MHz System clock while the EMIF is the same as that of Figure 6.6 interfacing with the common DDR3. Each Master Template imports/exports the signals contained in Table 6.1.

to extract data. All three Master Template cores can issue requests simultaneously, but the EMIF can only process one request at a time.

User Logic

The work done so far sets up different systems within the FPGA. Code is developed to harvest the use of each system. Software code running on the NIOS II processor starts up the Ethernet link and the NIOS II processor initiates the transfer of data to the DDR3. Once this is done, the exported signal `status_output` (Figure 6.7) is pulsed from low-to-high. This indicates that the processor is no longer using the DDR3.

The first step of the user logic is to track the value of the `status_output`. When a low-to-high pulse is detected, each of the Master Template IPs will be started i.e. asserting `control_go`. Also at this point, `status_input` will be asserted (high), indicating to the NIOS II processor not to use the DDR3.

Individual Master Template IPs read their respective 512 bits at different rates. Since the reads are sequential, the total read of 1536 bits can only be made once all 3 IPs return a valid 192-bit read. Figure 6.12 traces out the path after extraction of data from the DDR3.

Once the full 1536 bit register is ready, it is pushed into a **Dual-Clock FIFO (DCFIFO)** buffer. The DCFIFO is needed because of its ability to have different input and output widths. With respect to the lpGBT payload, the Injector only needs 192 bits at a time. An input wire of 1536 bits and an output wire of 192 bits is a legal input/output width ratio of 8, and can thus be implemented by a DCFIFO. The input and the output of the FIFO are both clocked by the same 80 MHz clock. 1536 bits are pushed every time all 3 IPs complete a read, and 192 bits are popped every 25 ns i.e. 40 MHz.

Extraction of 12×16-bit **ADCs** (192-bits) at 40 MHz is now achieved. However, the lpGBT payload also consists of 2×16-bit **BCID** bits (see Figure 5.1). To account for this, 2×16-bit registers are concatenated with the 192 data bits every time a data entry comes out from the DCFIFO. This now results in new 224 bits containing ADC+BCID information every 25 ns.

The 230 user-bits of the lpGBT payload contain 192-bits of ADC, 32-bits of BCIDs and 6 reserved bits. The function of the reserved bits has not been assigned yet and can be treated as zeros for the purpose of the Injector. Therefore, the data containing 224 user-bits of the lpGBT payload is now packaged and ready to be transferred to the LASP.

6.5.2 Data Transfer

In typical HL-LHC operation, FEB2 data is serialized and sent to the LASP using **lpGBT** chips at 10.24 Gbps. Since the lpGBT chips are still under production, a lpGBT emulator that can be programmed into hardware has been developed by CERN.

lpGBT emulator

The LASP implementation of the lpGBT, as shown in Figure 6.11, consists of

- **A Multi Gigabit Transceiver (MGT)** using Intel FPGA IP;
- Gearboxes, data-paths, and a frame aligner.

and sends/receives

- Downlink data (from LASP to FEB2) - runs at 2.56 Gbps:
  - User data (32-bit): sent to LpGBT e-links;
  - EC (2-bit): used for the external slow control;
  - IC (2-bit): used for the internal slow control of the lpGBT.
- Uplink data (from FEB2 to LASP) - runs at 10.24 Gbps:
  - Slow control (4-bit): IC (2-bit) and EC (2-bit);
  - User bandwith (230-bit): From LpGBT e-links (6-bit unconnected).

The Injector project is mainly concerned with the Uplink data path: From the FEB2 to the LASP. The uplink data-path (consisting of scramblers, encoders and interleavers) receives a 230-bit frame as well as IC and EC channels, and produces a 256-bit frame which is sent to the tx gearbox. The tx gearbox then splits the 256-bit input frame into 8 32-bit output frames and sends it to the MGT.

The MGT is implemented by Intel Stratix 10 FPGA Transceiver IPs which control the sending of data through a **QSFP** cable.

In order to achieve the transmission speeds needed, the transceivers (MGT) and the whole lpGBT emulator components run at 320 MHz.

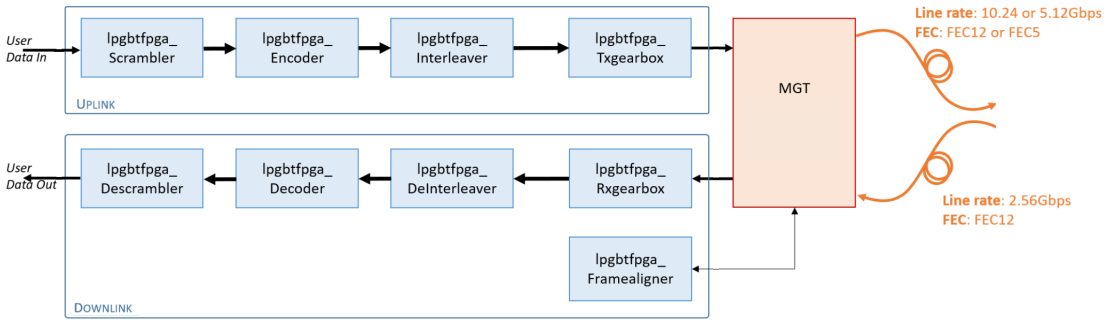


Figure 6.11: The lpGBT emulator module description. For the Injector, only the uplink data-path is used. Data passes through and is manipulated in the uplink path and sent using the MGT to the LASP via QSFP cables. On the LASP, a similar instance of the lpGBT is instantiated, although this time in reverse: the uplink and downlink data-paths are exchanged. Diagram taken from [47].

Interfacing with the lpGBT emulator

The lpGBT emulator interface and its components are clocked at 320 MHz. So far, the 224-bit payload (ADC+BCID) and the NIOS system all run at 80 MHz. Since these two regimes have different clocking domains, steps to avoid metastability must be considered.

In general, the output of an edge-triggered **flipflop** has two valid states: high and low. To ensure reliable operation, designs must meet the flipflop's timing requirements. The input to the flipflop must be stable for a minimum time before the clock edge (register setup time or t_{SU}) and a minimum time after the clock edge (register hold time or t_H). When dealing with different clock domains, a flipflop's setup or hold time can be violated causing its output to become **metastable** [48]. A metastable state is one in which the output of a flipflop inside the FPGA is unknown, or non-deterministic. This causes unpredictable FPGA behaviour and system failure.

To ensure that the different clock domains are crossed properly, the design uses another Dual Clock FIFO, DCFIFO#2 (#2 differentiates it from the DCFIFO used prior). The DCFIFO has an input end and an output end that can be clocked by different oscillators. Using a cascade of flipflops within the FIFO, the DCFIFO ensures that even though the middle layer flipflops may be metastable, the last flipflop is not.

For our implementation, the 224-bit register is fed into the input side of DCFIFO#2 every 25 ns. The input side is clocked at 80 MHz. The output of DCFIFO#2 is a 230-bit register containing the 224 data bits concatenated with 6 empty bits (zeros). This 230-bit payload is popped out every 25 ns as well. The output side of the

DCFIFO is clocked at 320 MHz. This ensures that the 230-bit payload received at the output end of the DCFIFO#2 is stable in the 320 MHz clock regime.

The 230-bit payload is then connected to the input end of the lpGBT_emulator uplink data path. The lpGBT_emulator packages the payload according to the lpGBT protocol and sends it to the LASP through the functionality of the MGT. In the LASP, the instance of the lpGBT is set up to receive and descramble/decode the 230-bits received.

6.5.3 Summary

The summary of the Data retrieval and transfer stage is shown in Figure 6.12.

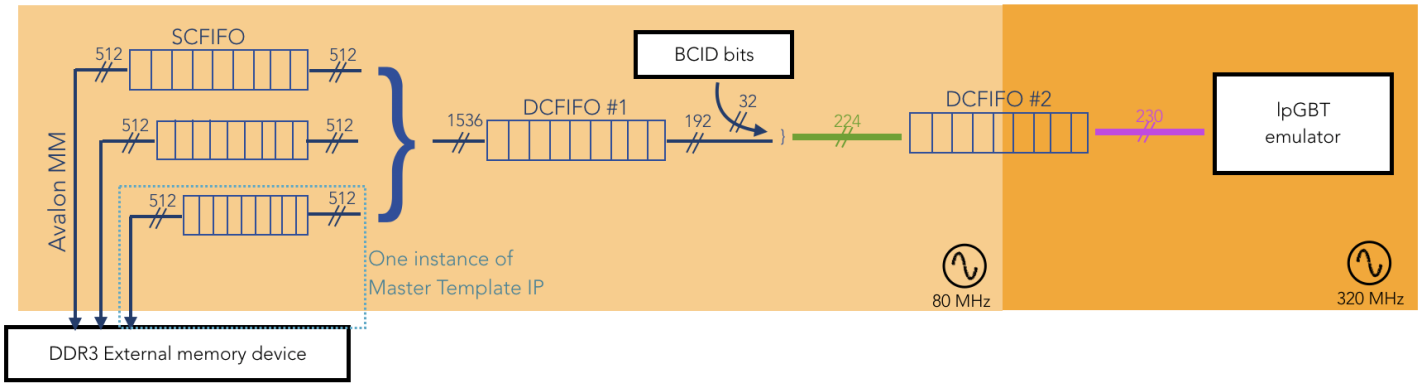


Figure 6.12: Block diagram showing the design flow of the data extraction and transfer stage. SCFIFO represents FIFOs that are clocked by a Single Clock. The different shade of colours represent the different clock regimes the logic resides in.

Three instances each of Master Template IP are used to extract 512 bits from the DDR3. The Master Template IP uses a regular **Single-clock FIFO (SCFIFO)** to present data to user logic. The output from the 3 SCFIFOs are concatenated and pushed into DCFIFO#1. 192 bits of ADC data are popped out at the output end of this FIFO at 40 MHz i.e. every 25 ns. This 192 bits are then concatenated with 2×16-bit BCIDs. The result of this concatenation is pushed every 25 ns into DCFIFO#2 whose output is a 230-bit register which is stable in 320 MHz clock regime. DCFIFO#2 is popped every 25 ns as well. This payload is fed to the lpGBT_emulator which packages and sends it to the LASP.

The read from the DDR3 continues indefinitely. Once the full memory length has been read, the Master Template IP is issued another `control_go` command, and the whole Data Retrieval and Transfer process above is repeated.

Chapter 7

Injector Performance and Results

The major criterion for measuring the functionality of the Injector project is a pass/-fail determination of whether it can receive, hold, and inject simulated **FEB2** pulses to the **LASP**. A test to confirm that the Injector project meets this minimum requirement is presented in 7.1. The following extra criteria are used to evaluate the performance of the Injector.

- Transmission Accuracy - Results of this subject is discussed in section 7.2
- Quantity of Simulated FEB2 data transferred - Results of this yardstick is discussed in section 7.3

Other notes about the Injector project concern how much of the FPGA resources the project uses. This is discussed in section 7.4.

7.1 Basic Injector Functionality

The primary role of the Injector is three fold: To receive data from a workstation, store that data in an external memory device, and to transfer that data to the LASP. An acid test for this is to use known data from the workstation and to monitor the output of the payload received in the LASP. If the payload is the same as the data the workstation sent, then the Injector project is deemed functional.

7.1.1 Nature of Data

In normal operation, simulated FEB2 pulses would be sent from the workstation. Because of the randomness in pileup, the readout pulses would be an irregular stream

of ADC values. These could be unintuitive to match to the data received in the LASP.

For this reason, to verify and validate the functionality of the Injector project, we use a regular counter as the data to be sent. `bulk_client_send.C` (Figure 6.8) is the program that is executed on the workstation. This program sends incrementing 16-bit numbers (`shorts`) starting from 1.

The current version of `bulk_client_send.C` sends 966,367,641 numbers. This amounts to sending 1.8 GB of data. The `unsigned short` type consists of the range [0,65535]. Thus every 65,535 times, the counter resets back to 0.

The **NIOS II** processor receives these 16-bit unsigned shorts (aliased as `alt_u16` in NIOS II FPGAs) and stores the bits, in the order they are received, into the DDR3.

Once all the storage is done, the Injector then retrieves these bits and sends them to the LASP.

7.1.2 Successful Data Transfer from workstation to Injector

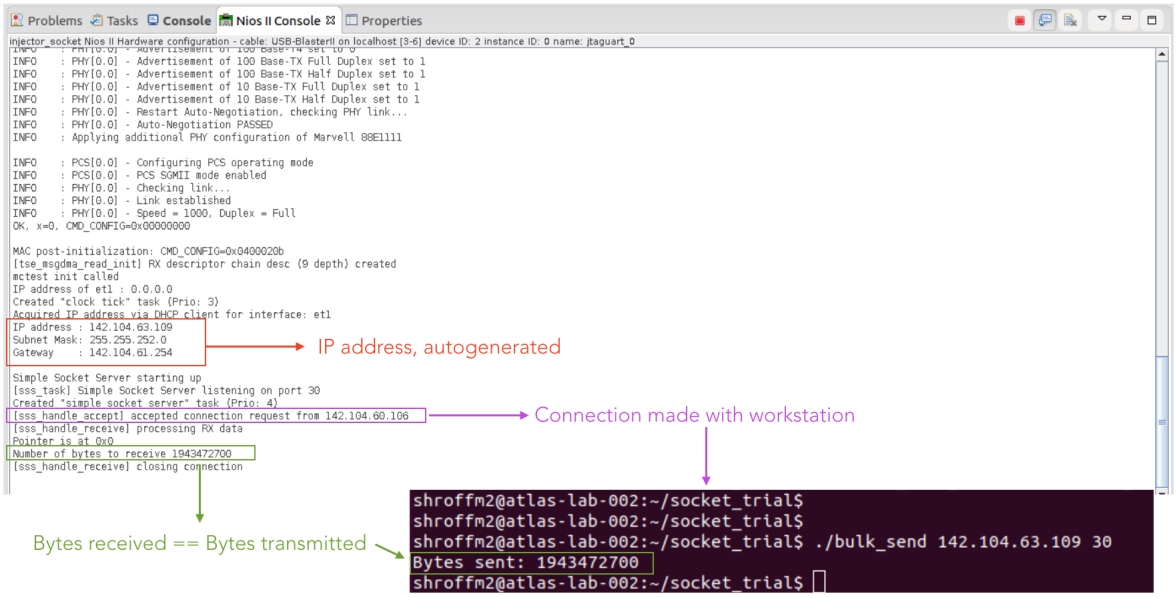


Figure 7.1: Output of the NIOS II processor indicating the status of the Ethernet Stack setup as well as the socket creation. Insert: Output of running `bulk_send`. Note the number of bytes sent are the same as the number of bytes received.

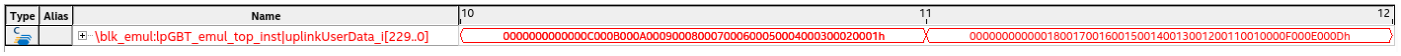
The startup of the Ethernet on the Injector configured FPGA can be seen via a **JTAG UART** connection through which the NIOS II processor is configured. Figure 7.1 shows the successful implementation of a **socket** server. The IP address

is then used as an argument to the executable `bulk_send` (which is created from `bulk_client_send.C`) which sends the counter data to the FPGA. Once all the bytes have been sent, the client closes the socket connection.

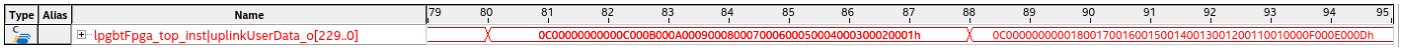
7.1.3 Monitoring the LASP

Values of signals inside the FPGA can be monitored while the device is running using a **Quartus** tool called **SignalTap**. This tool allows us to check the LASP wire that receives the payload. Similar wires can also be tapped in the Injector to ensure that the reads from DDR3 and the payload to be sent are prepared correctly.

The first two values of the signal being sent from the Injector project is shown in Figure 7.2a. The corresponding signal being received in the LASP is shown in Figure 7.2b.



(a) Outgoing payload in the Injector



(b) Incoming payload in the LASP

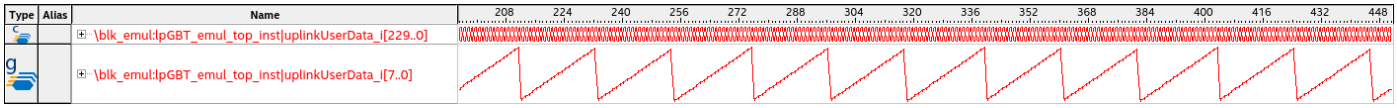
Figure 7.2: SignalTap instances of the first two payloads in the Injector and the LASP. The clock used for the SignalTap instance in (a) is the 40 MHz derivative of the MGT. The clock used for the SignalTap instance in (b) is the 320 MHz MGT clock. The payload in (b) changes its value after 8 clock ticks.

Since the payload is 230 bits wide, the output for a number of payloads is hard to capture at one instance. For visualisation of how the bits change, focus is given to the first 8 bits of every payload and its evolution over a period of time is monitored. The value of the 8-bits can be represented as values in an unsigned line graph. This is shown in Figure 7.3.

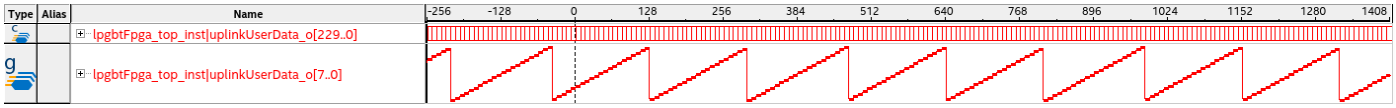
The line graph for the 8 bits shows an increasing counter which resets periodically. The counter can be seen in both the receiving end and transceiving end. This proves basic Injector functionality.

7.2 Transmission Accuracy

Transmission Accuracy is the measure that investigates the number of differences between the payload that is being sent from the Injector and the payload that is being



(a) Outgoing payload in the Injector



(b) Incoming payload in the LASP

Figure 7.3: SignalTap instances of the first 8 bits in every payload in the Injector and the LASP. The clock used for the SignalTap instance in (a) is the 40 MHz derivative of the MGT. The clock used for the SignalTap instance in (b) is the 320 MHz MGT clock.

received by the LASP. It is a stringent test that also investigates the functionality of the lpGBT emulator, **MGT** and, **QSFP** cables.

To measure the transmission accuracy, the **BCID** bits on the payload are replaced to incorporate a **checksum.** The checksum will be calculated within the Injector project and sent with the lpGBT payload. Upon receipt in the LASP, the payload will be further manipulated to measure if the payload has changed during transmission.

7.2.1 Longitudinal Parity Checksum

A **Longitudinal Parity Checksum (LPC)** breaks the data into n-bit "bunches" (words) and computes the Exclusive OR (XOR) of all of these words. The result is one n-bit word that is appended to the payload as an extra word. At the receiving end, an XOR is performed of the whole payload, including the checksum, in n-bit words. If the result is not a word consisting of n zeros, a transmission error has occurred.

In the Injector Project, the payload is broken down into 6×32 bit words, which primarily contain **ADC** data. The XOR computed on these words results in one 32-bit word, which is substituted to take the place of the 2×16-bit BCID bits. This makes up a "modified" 224-bit user payload. If the transmission is accurate, we can be assured that all 224-bits are correctly transmitted to the LASP. In normal operation, this would mean that the ADCs and the BCIDs will be transmitted correctly. A diagrammatic representation of how the LPC is incorporated into the payload is shown in Figure 7.4.

The LASP is also modified to then process the payload to check for errors. It does this by computing a simple XOR of the 224-bits in 32-bit bunches. If the checksum

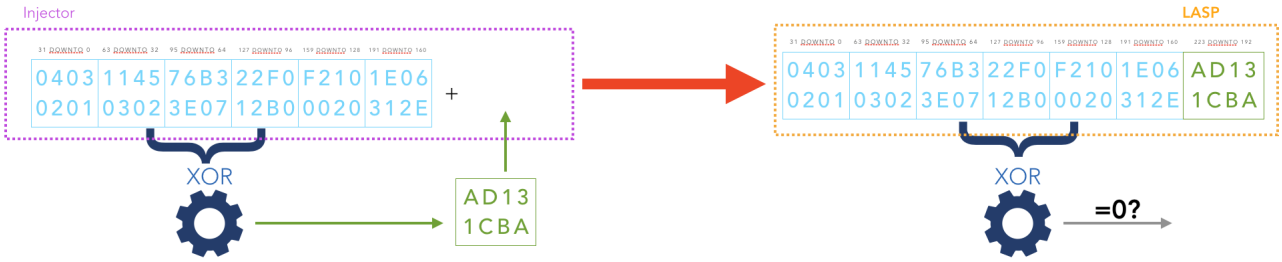


Figure 7.4: Methodology of adding the Checksum (`AD131CBA` in Figure) to the Injector Payload. The payload is represented in Hexadecimals. 1 Hexadecimal equates to 4 bits

and the 6 ADC words are transmitted accurately, the result of the XOR would be 0. For every payload received, if the result of the XOR is not 0, a counter that keeps track of the number of errors detected is incremented.

Limitations of LPC

The LPC is a simple and quick way to check the integrity of a payload, but comes with some limitations. The LPC will function correctly and will detect an error if single bits, or an odd number of bits, are flipped in the payload. However, an error which affects two bits will not be detected if those bits lie at the same position in two distinct words. Further, the LPC will fail to detect the swapping of two or more words. If the affected bits are independently chosen at random, the probability of a two-bit error being undetected is $\frac{1}{32} = 3.125\%$.

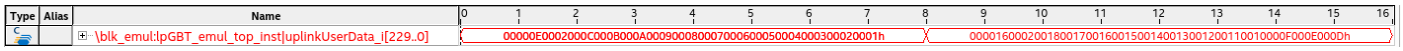
7.2.2 Transmission Accuracy Results

Figure 7.5 illustrates the transceiving and receiving payload with the checksum incorporated. Also the value of the error counter, `error_count`, and the XOR result on each received payload, `xor_result`, is monitored.

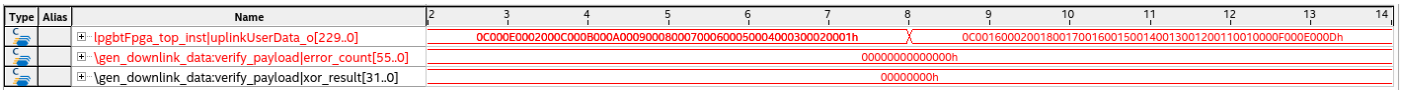
Continued long-standing tests of running the Injector project has shown to have a 100% transmission accuracy. This can be attributed to multiple factors:

- The secure nature of the TCP/IP protocol ensures that every packet transmitted is received on the other end. This ensures a reliable communications link between the workstation and the FPGA

- The FPGA design meets all timing requirements. This means that the integrity



(a) Outgoing payload in the Injector



(b) Incoming payload in the LASP

Figure 7.5: SignalTap instances of the first two payloads with the checksum bits. `error_count` in (b) measures the number of times `xor_result` $\neq 0$, which means a transmission error has occurred.

of the design will continue to run as planned, up until the point of hardware failure (loss of power, overheating, etc)

- The lpGBT protocol consists of FEC (Forward Error Control) bits that can correct up to 5 consecutive errors. This sophisticated error correction ensures an accurate transmission from Injector to LASP.

7.3 Quantity of data transferred

The quantity of data that can be stored at any one time is limited to the capacity of the external DDR3 memory chip. Currently, a 2 GB chip is used. This means that at most 2 GB of data can be transferred from the Ethernet to the FPGA at one time.

The throughput of the Ethernet and consequently the rate at which data is written into the DDR3 is $\sim$ 2.7 MBps. This is in stark contrast to the rate at which data can be read from the DDR3: 4 GBps. Further, the throughput required for the LAPS is 960 MBps. Because of the massive lag in the Ethernet throughput, the DDR3 cannot be updated on-the-fly. Thus, a maximum of 2 GB can be transferred to the LASP. The current Injector project is implemented to read 1.8 GB from the DDR3 (this parameter can be changed).

The Ethernet throughput limitation results from a slow NIOS II clock speed (80 MHz) and the simple implementation of a TCP/IP stack. Because the processor follows a RISC architecture, it is unreasonable to expect the Ethernet throughput to be $\geq$ 100 MBps.

A capacity of 1.8 GB would mean having 966,367,641 ADC values for one single LAr Calorimeter cell. This translates to having 12 seconds of both high gain and low gain ADCs for one calorimeter cell.

ADCs for a block of Calorimeter cells could also be used. For a 3×2 array in η and ϕ, respectively, and storing 2 gains for each BC per cell, the Injector can store data for $\sim$80 million bunch crossings i.e 2 seconds of calorimeter operation.

It is important to note that the Injector project is designed to run indefinitely. After a complete 1.8 GB read, the Injector will loop-back to the starting address and continue to complete consequent 1.8 GB reads and transmit the data. The memory capacity only offers a limitation to the size of unique data that can be stored. With respect to the LASP, it would continue receiving data as expected indefinitely.

7.4 FPGA resource usage

A good-to-have criteria for the Injector project is that it would need to be light-weight with respect to the amount of FPGA resources it uses. This is because, although the Injector is designed to be programmed onto its own separate FPGA devkit, it can also be programmed into the LASP itself. In such a mode, the Injection can be done using the transceivers and receivers internal loop-back capabilities. Thus, a light-weight design is a plus as it can then be combined with other modules of the test-bench. Table 7.1 illustrates the resources used by a typical implementation of the Injector.

Family	Stratix 10
Device	1SG280LU2F50E2VGS3
Logic Utilization (in ALMs)	25,083 / 933,120 (3%)
Total dedicated logic registers	52030
Total pins	195 / 1,152 (17%)
Total block memory bits	18,847,000 / 240,046,080 (8%)
Total DSP Blocks	2 / 5,760 (<1%)
Total PLLs	6 / 184 (3%)

Table 7.1: Summary of the resources used in a typical implementation of the Injector project.

DSPs are valuable resources for running Digital Filters to process the pulses in the LASP. The Injector usage of DSPs is minimal.

An important usage for the Injector project is the total block memory bits, which is under 10% of the total FPGA capacity. The largest block of memory implemented is the 2 MB on-chip memory for the NIOS II. The **FIFOs** occupy $\leq$ 100 KB.

7.5 Summary of Results

The results are summarized in Table 7.2 below.

Criteria	Performance
Basic functionality	✓
Amount of unique data stored	1.8 GB[1]
Transmission Accuracy	No observed errors
Ethernet throughput	∼2.7 MBps
Total memory bits usage	8%
Duration of Injection	≥ 5 days
Number of transmission channels	4

Table 7.2: Summary of the Injector project performance

[1]The true theoretical maximum is ∼2 GB. The current design uses 1.8 GB

Chapter 8

Conclusions and Outlook

The 2012 discovery of the Higgs Boson in the **LHC** [6] completed the search for the last missing particle in the Standard Model prediction. So far, no deviations from the Standard Model have been experimentally verified. However, present measurement precision still does not exclude certain Beyond the Standard Model (BSM) theories. In an attempt to increase the precision of measurements, an upgraded version - **"HL-LHC"** of the collider with a larger instantaneous luminosity is planned. This upgrade would mean the replacement of existing components, such as the **LAr Calorimeter** electronics in the **ATLAS** detector [24], to work more efficiently and reliably in a high luminosity environment.

The LAr Calorimeter will see changes to both its front-end and off-detector electronics. On the front-end, new **FEB2** boards will be installed that would preamplify, shape, and digitize two gains of the detector signal into 14-bit ADCs every 40 MHz. The **ADCs** are then serialized and sent to the **LASP** at 10.24 Gbps using the **lpGBT** protocol [24]. The off-detector LASP receives the FEB2 data and processes it. This involves, amongst other tasks, applying digital filters to calculate the energy/time of the signal pulses. The LASP then sends the information to different trigger systems whilst buffering the data, until a trigger decision is reached. Because of the complexity of LASP requirements, **FPGA** technologies are used to develop a prototype of the LASP [24] through which algorithms can be changed in an ad-hoc manner.

In this thesis, an Injector has been developed as a test-bench for the LASP. Because FEB2 pulses will not be available until HL-LHC operation, the Injector project transmits simulated pulses to the LASP. This helps develop and test different LASP modules. The simulated FEB2 pulses are obtained from a Monte-Carlo simulation program called **AREUS**. The Injector is designed on a Stratix 10 FPGA Development

kit - similar to that of the LASP prototype.

The Injector is designed such that it connects to a host PC via an **Ethernet** link. The Ethernet on the FPGA is controlled by a soft-core **NIOS II** processor which sets up a **TCP/IP socket** server. The host PC sends AREUS/user-defined data to the FPGA. Because of low throughput, data received from the host PC cannot directly be sent to the LASP, but instead is stored in an External **DDR3** memory chip on the Injector FPGA board. Once 1.8 GB of data is received and stored in the DDR3 device, its contents are then transmitted to the LASP at 10.24 Gbps.

Tests on the full working Injector module show a transmission accuracy of 100%. A major limiting factor is the low Ethernet throughput of ∼2.7 MBps. This is attributed to a slow processor clock as well as an implementation of a simple TCP/IP stack. With this limitation, the Injector can transmit 966,367,641 unique ADC values if only one single LAr Calorimeter cell is considered. If all the calorimeter cells are considered, the Injector can transmit 5,300 unique ADC values.

Improvement of the Injector performance would be possible by modifying the communication mechanism from host PC to Injector FPGA. Dedicated communication hardware can be used to replace soft-core processor controlled Ethernet. One such example is the **FELIX** system, which is employed in transmission of data from the LASP to the **TTC** systems in the HL-LHC [24]. The FELIX FPGA **PCIe** card (FLX-712) has transmission capability that exceeds the LASP requirement of 1.15 GBps (230 user bits in 25 ns) [30]. Details of its implementation is still to be explored by the group as the FELIX prototypes become available for use.

Appendix A

Injector VHDL Code

```
1  ----------------------------------------------------------------------------
2  -- EMACS settings: -*- tab-width: 2; indent-tabs-mode: nil -*-
3  -- vim: tabstop=2:shiftwidth=2:expandtab
4  -- kate: tab-width 2; replace-tabs on; indent-width 2;
5  ----------------------------------------------------------------------------
6  --! @file
7  --! @author Maheyer J, Shroff <maheyer.jamshed.shroff@cern.ch>
8  --! @brief  An entity that can receives user defined data from a workstation
9  --! and prepares lpGBT payloads using this data
10 --!
11 ----------------------------------------------------------------------------
12 --! @details This entity instantiates a Platform designer system, BUP_QSYS, which
13 --! involves a NIOS II processor, a Triple Speed Ethernet (TSE) IP and ability
14 --! to access data within an external DDR3 memory device using an EMIP IP.
15 --! The data is then read and packaged in the lpGBT payload structure.
16 --!
17 --! @section inst_inj Instantiation template:
18 --!
19 --! @code{.vhdl}
20 --! [inst_name]: entity injector.injector
21 --!  port map (
22 --!    clk_320                => [in std_logic],                      -- MGT clock
23 --!    clk_133                => [in std_logic],                      -- EMIF clock
24 --!    clk_125                => [in std_logic],                      -- Ethernet clock
25 --!    cpu_resetn1            => [in std_logic],                      -- Active low reset
26 --!    enable                 => [in std_logic],                      -- Transmitter strobe (1/8 of clock)
27 --!    payload                => [out std_logic_vector(229 downto 0)], -- Uplink Data prepared
28 --!    mem_a1                 => [out std_logic_vector(14 downto 0)], -- EMIF Address
29 --!    mem_ba1                => [out std_logic_vector(2 downto 0)],  -- EMIF Bank Address
30 --!    mem_ck1                => [out std_logic],                     -- EMIF CK clock
31 --!    mem_ck_n1              => [out std_logic],                     -- EMIF CK clock (negative leg)
32 --!    mem_cke1               => [out std_logic],                     -- EMIF clock enable
33 --!    mem_dqs1               => [inout std_logic_vector(8 downto 0)], -- EMIF data strobe
34 --!    mem_dqs_n1             => [inout std_logic_vector(8 downto 0)], -- EMIF data strobe (negative leg)
35 --!    mem_dq1                => [inout std_logic_vector(71 downto 0)], -- EMIF Read/Write data
36 --!    mem_cs_n1              => [out std_logic],                     -- EMIF chip select
37 --!    mem_reset_n1           => [out std_logic],                     -- EMIF Asynchronous reset
38 --!    mem_odt1               => [out std_logic],                     -- EMIF On-die termination
39 --!    mem_we_n1              => [out std_logic],                     -- EMIF WE command
40 --!    mem_ras_n1             => [out std_logic],                     -- EMIF RAS command
41 --!    mem_cas_n1             => [out std_logic],                     -- EMIF CAS command
42 --!    mem_dm1                => [out std_logic_vector(8 downto 0)],  -- EMIF Write data mask
43 --!    oct_rzqin1             => [in std_logic],                      -- EMIF Calibrated On-Chip Termination RZQ
44 --!    enet_sgmii_tx_p        => [out std_logic],                     -- Ethernet SGMII transmit channel
45 --!    enet_sgmii_rx_p        => [in std_logic],                      -- Ethernet SGMII receive channel
46 --!    enet_rstn              => [out std_logic],                     -- Ethernet Device reset
47 --!    enet_intn              => [in std_logic],                      -- Ethernet Management bus interrupt
48 --!    enet_mdc1              => [out std_logic],                     -- Ethernet Management bus data clock
49 --!    enet_mdio1             => [inout std_logic]                    -- Ethernet Management bus data
```

```
50  --!  );
51  --! @endcode
52  ------------------------------------------------------------------------------
53  --! @cond
54  library IEEE;
55  context IEEE.IEEE_STD_CONTEXT;
56
57  library PoC;
58    use PoC.utils.all;
59
60  library misc;
61  --! @endcond
62
63  entity injector is
64    port (
65      clk_320                 : in    std_logic;                          --! MGT clock
66      clk_133                 : in    std_logic;                          --! EMIF clock
67      clk_125                 : in    std_logic;                          --! Ethernet clock
68      cpu_resetn1             : in    std_logic;                          --! System reset
69      enable                  : in    std_logic;                          --! Transmit strobe
70      payload                 : out   std_logic_vector(229 downto 0);     --! Uplink payload
71      mem_a1                  : out   std_logic_vector(14 downto 0);      --! EMIF Address
72      mem_ba1                 : out   std_logic_vector(2 downto 0);       --! EMIF Bank Address
73      mem_ck1                 : out   std_logic;                          --! EMIF CK clock
74      mem_ck_n1               : out   std_logic;                          --! EMIF CK clock inverted
75      mem_cke1                : out   std_logic;                          --! EMIF clock enable
76      mem_dqs1                : inout std_logic_vector(8 downto 0);       --! EMIF data strobe
77      mem_dqs_n1              : inout std_logic_vector(8 downto 0);       --! EMIF data strobe inverted
78      mem_dq1                 : inout std_logic_vector(71 downto 0);      --! EMIF read/write data
79      mem_cs_n1               : out   std_logic;                          --! EMIF chip select
80      mem_reset_n1            : out   std_logic;                          --! EMIF Asynchronous reset
81      mem_odt1                : out   std_logic;                          --! EMIF On-die termination
82      mem_we_n1               : out   std_logic;                          --! EMIF WE command
83      mem_ras_n1              : out   std_logic;                          --! EMIF RAS command
84      mem_cas_n1              : out   std_logic;                          --! EMIF CAS command
85      mem_dm1                 : out   std_logic_vector(8 downto 0);       --! EMIF Write data mask
86      oct_rzqin1              : in    std_logic;                          --! EMIF Calibrated On-Chip Termination RZQ
87      enet_sgmii_tx_p         : out   std_logic;                          --! Ethernet SGMII transmit channel
88      enet_sgmii_rx_p         : in    std_logic;                          --! Ethernet SGMII receive channel
89      enet_rstn               : out   std_logic;                          --! Ethernet Device reset
90      enet_intn               : in    std_logic;                          --! Ethernet Management bus interrupt
91      enet_mdc1               : out   std_logic;                          --! Ethernet Management bus data clock
92      enet_mdio1              : inout std_logic                           --! Ethernet Management bus data
93    );
94  end injector;
95
96  --! Implementation of injector
97
98  architecture behavioral of injector is
99
100   --! signals dealing with the Ethernet controls/pins
101   signal mdio_in                          : std_logic;
102   signal mdio_oen                         : std_logic;
103   signal mdio_out                         : std_logic;
104   signal enet_resetn                      : std_logic;
105
106   --! status signals
107   signal status_wire_in                   : std_logic;
108   signal status_wire_out                  : std_logic;
109   signal go                               : std_logic;
110   signal read_all                         : std_logic;
111   signal control_done_0                   : std_logic;
112   signal control_done_1                   : std_logic;
113   signal control_done_2                   : std_logic;
114   signal data_available_0                 : std_logic;
115   signal data_available_1                 : std_logic;
116   signal data_available_2                 : std_logic;
117   signal read_enable                      : std_logic                 := '0';
118   signal first_read                       : std_logic;
119   signal fifo_1_full                      : std_logic;
120   signal fifo_1_empty                     : std_logic;
```

```
121     signal fifo_1_empty_delay                   : std_logic;
122     signal transfer_complete                    : std_logic;
123     signal fifo_2_empty                         : std_logic;
124
125     --! Data reads
126     signal obtained_data_0                      : std_logic_vector(511 downto 0);
127     signal obtained_data_1                      : std_logic_vector(511 downto 0);
128     signal obtained_data_2                      : std_logic_vector(511 downto 0);
129     signal full_read                            : std_logic_vector(1535 downto 0);
130     signal master_buffer_0                      : std_logic_vector(511 downto 0);
131     signal master_buffer_1                      : std_logic_vector(511 downto 0);
132     signal master_buffer_2                      : std_logic_vector(511 downto 0);
133     signal payload_nocheck                      : std_logic_vector(191 downto 0);
134     signal payload_checksum                     : std_logic_vector(223 downto 0);
135     signal payload_320m                         : std_logic_vector(229 downto 0);
136
137     --! Checksum signal
138     signal check_sum                            : std_logic_vector(31 downto 0);
139
140     --! Intermediate dummy signals
141     signal dummy                                : unsigned(191 downto 0)        := (others => '1');
142
143     --! Clock derivatives, slow control signals annd BC frequency signals.
144     signal clk_counter                          : std_logic_vector(1 downto 0)  := (others => '0');
145     signal clk_80m                              : std_logic;
146     signal global_resetn                        : std_logic;
147     signal enable_40m                           : std_logic;
148     signal slow_clk                             : std_logic;
149     signal slow_clk_count                       : std_logic;
150     signal reset1_counter                       : std_logic;
151     signal reset2_counter                       : std_logic;
152
153     --! Declaration of the Platform Deisgner/QSYS system
154
155     component BUP_QSYS is
156       port (
157         clk_125m_clk                            : in    std_logic                       := '0';
158         reset_125m_reset_n                      : in    std_logic                       := '0';
159         clk_133m_clk                            : in    std_logic                       := '0';
160         local_reset_req_local_reset_req         : in    std_logic                       := '0';
161         local_reset_status_local_reset_done     : out   std_logic;
162         mem_mem_ck                              : out   std_logic;
163         mem_mem_ck_n                            : out   std_logic;
164         mem_mem_a                               : out   std_logic_vector(14 downto 0);
165         mem_mem_ba                              : out   std_logic_vector(2 downto 0);
166         mem_mem_cke                             : out   std_logic;
167         mem_mem_cs_n                            : out   std_logic;
168         mem_mem_odt                             : out   std_logic;
169         mem_mem_reset_n                         : out   std_logic;
170         mem_mem_we_n                            : out   std_logic;
171         mem_mem_ras_n                           : out   std_logic;
172         mem_mem_cas_n                           : out   std_logic;
173         mem_mem_dqs                             : inout std_logic_vector(8 downto 0)    := (others => '0');
174         mem_mem_dqs_n                           : inout std_logic_vector(8 downto 0)    := (others => '0');
175         mem_mem_dq                              : inout std_logic_vector(71 downto 0)   := (others => '0');
176         mem_mem_dm                              : out   std_logic_vector(8 downto 0);
177         oct_oct_rzqin                           : in    std_logic                       := '0';
178         status_local_cal_success                : out   std_logic;
179         status_local_cal_fail                   : out   std_logic;
180         master_template_0_control_fixed_location  : in    std_logic                     := '0';
181         master_template_0_control_read_base       : in    std_logic_vector(31 downto 0) := (others => '0');
182         master_template_0_control_read_length     : in    std_logic_vector(31 downto 0) := (others => '0');
183         master_template_0_control_go              : in    std_logic                     := '0';
184         master_template_0_control_done            : out   std_logic;
185         master_template_0_control_early_done      : out   std_logic;
186         master_template_0_user_read_buffer        : in    std_logic                     := '0';
187         master_template_0_user_buffer_output_data : out   std_logic_vector(511 downto 0);
188         master_template_0_user_data_available     : out   std_logic;
189         master_template_1_control_fixed_location  : in    std_logic                     := '0';
190         master_template_1_control_read_base       : in    std_logic_vector(31 downto 0) := (others => '0');
191         master_template_1_control_read_length     : in    std_logic_vector(31 downto 0) := (others => '0');
```

```
192         master_template_1_control_go              : in    std_logic                     := '0';
193         master_template_1_control_done            : out   std_logic;
194         master_template_1_control_early_done      : out   std_logic;
195         master_template_1_user_read_buffer        : in    std_logic                     := '0';
196         master_template_1_user_buffer_output_data : out   std_logic_vector(511 downto 0);
197         master_template_1_user_data_available     : out   std_logic;
198         master_template_2_control_fixed_location  : in    std_logic;
199         master_template_2_control_read_base       : in    std_logic_vector(31 downto 0)  := (others => '0');
200         master_template_2_control_read_length     : in    std_logic_vector(31 downto 0)  := (others => '0');
201         master_template_2_control_go              : in    std_logic                      := '0';
202         master_template_2_control_done            : out   std_logic;
203         master_template_2_control_early_done      : out   std_logic;
204         master_template_2_user_read_buffer        : in    std_logic                      := '0';
205         master_template_2_user_buffer_output_data : out   std_logic_vector(511 downto 0);
206         master_template_2_user_data_available     : out   std_logic;
207         status_input_external_connection_export   : in    std_logic                      := '0';
208         status_output_external_connection_export  : out   std_logic;
209         sys_clk_clk                               : in    std_logic                      := '0';
210         reset_1_reset_n                           : in    std_logic                      := '0';
211         tse_mac_mac_mdio_connection_mdc           : out   std_logic;
212         tse_mac_mac_mdio_connection_mdio_in       : in    std_logic                      := '0';
213         tse_mac_mac_mdio_connection_mdio_out      : out   std_logic;
214         tse_mac_mac_mdio_connection_mdio_oen      : out   std_logic;
215         tse_mac_misc_connection_magic_wakeup      : out   std_logic;
216         tse_mac_misc_connection_magic_sleep_n     : in    std_logic                      := '0';
217         tse_mac_misc_connection_ff_tx_crc_fwd     : in    std_logic                      := '0';
218         tse_mac_misc_connection_ff_tx_septy       : out   std_logic;
219         tse_mac_misc_connection_tx_ff_uflow       : out   std_logic;
220         tse_mac_misc_connection_ff_tx_a_full      : out   std_logic;
221         tse_mac_misc_connection_ff_tx_a_empty     : out   std_logic;
222         tse_mac_misc_connection_rx_err_stat       : out   std_logic_vector(17 downto 0);
223         tse_mac_misc_connection_rx_frm_type       : out   std_logic_vector(3 downto 0);
224         tse_mac_misc_connection_ff_rx_dsav        : out   std_logic;
225         tse_mac_misc_connection_ff_rx_a_full      : out   std_logic;
226         tse_mac_misc_connection_ff_rx_a_empty     : out   std_logic;
227         tse_mac_serial_connection_rxp_0           : in    std_logic                      := '0';
228         tse_mac_serial_connection_txp_0           : out   std_logic;
229         tse_mac_status_led_connection_crs         : out   std_logic;
230         tse_mac_status_led_connection_link        : out   std_logic;
231         tse_mac_status_led_connection_panel_link  : out   std_logic;
232         tse_mac_status_led_connection_col         : out   std_logic;
233         tse_mac_status_led_connection_an          : out   std_logic;
234         tse_mac_status_led_connection_char_err    : out   std_logic;
235         tse_mac_status_led_connection_disp_err    : out   std_logic
236       );
237     end component;
238
239     --! Declaration of a DC-FIFO
240
241     component FIFO_DUALWIDTH is
242       port (
243         data    : in    std_logic_vector(1535 downto 0) := (others => '0');
244         wrreq   : in    std_logic                       := '0';
245         rdreq   : in    std_logic                       := '0';
246         wrclk   : in    std_logic                       := '0';
247         rdclk   : in    std_logic                       := '0';
248         q       : out   std_logic_vector(191 downto 0);
249         rdempty : out   std_logic;
250         wrfull  : out   std_logic
251       );
252     end component;
253
254     --! Declaration of a DC-FIFO
255
256     component CLK_CROSSING is
257       port (
258         data    : in    std_logic_vector(229 downto 0) := (others => '0');
259         wrreq   : in    std_logic                      := '0';
260         rdreq   : in    std_logic                      := '0';
261         wrclk   : in    std_logic                      := '0';
262         rdclk   : in    std_logic                      := '0';
```

```
263         q       : out   std_logic_vector(229 downto 0);
264         rdempty : out   std_logic;
265         wrfull  : out   std_logic
266       );
267     end component;
268 
269   begin
270 
271     --! Assignment of Ethernet intermediate wires
272     mdio_in    <= enet_mdio1;
273     enet_mdio1 <= mdio_out when (not mdio_oen) else 'Z';
274     enet_rstn  <= enet_resetn;
275 
276     --! Calculation of 32-bit checksum
277     check_sum  <= (payload_nocheck(31 downto 0) xor payload_nocheck(63 downto 32) xor payload_nocheck(95 downto 64) xor
      ↪  payload_nocheck(127 downto 96) xor payload_nocheck(159 downto 128) xor payload_nocheck(191 downto 160));
278 
279     --! Status wire instructing NIOS to stop using the DDR3
280     status_wire_in <= '1' when read_enable else '0';
281 
282     --! Preparation of final paylaod
283     payload   <= payload_320m when (not fifo_2_empty) else ((37 downto 0 => '0') & std_logic_vector(dummy));
284 
285     --! Instantiate the Platform Designer/ QSYS component
286     qsys_unit : bup_qsys
287       port map (
288         --! Resets, clocks
289         reset_1_reset_n                                 => global_resetn,
290         reset_125m_reset_n                              => global_resetn,
291         clk_125m_clk                                    => clk_125,
292         sys_clk_clk                                     => clk_80m,
293         clk_133m_clk                                    => clk_133,
294 
295         --! Ethernet conduits
296         tse_mac_mac_mdio_connection_mdc                 => enet_mdc1,
297         tse_mac_mac_mdio_connection_mdio_in             => mdio_in,
298         tse_mac_mac_mdio_connection_mdio_out            => mdio_out,
299         tse_mac_mac_mdio_connection_mdio_oen            => mdio_oen,
300         tse_mac_serial_connection_rxp_0                 => enet_sgmii_rx_p,
301         tse_mac_serial_connection_txp_0                 => enet_sgmii_tx_p,
302         tse_mac_misc_connection_magic_sleep_n           => '1',
303         tse_mac_misc_connection_ff_tx_crc_fwd           => '0',
304 
305         --! EMIF conduits
306         mem_mem_ck                                      => mem_ck1,
307         mem_mem_ck_n                                    => mem_ck_n1,
308         mem_mem_a                                       => mem_a1,
309         mem_mem_ba                                      => mem_ba1,
310         mem_mem_cke                                     => mem_cke1,
311         mem_mem_cs_n                                    => mem_cs_n1,
312         mem_mem_odt                                     => mem_odt1,
313         mem_mem_reset_n                                 => mem_reset_n1,
314         mem_mem_we_n                                    => mem_we_n1,
315         mem_mem_ras_n                                   => mem_ras_n1,
316         mem_mem_cas_n                                   => mem_cas_n1,
317         mem_mem_dqs                                     => mem_dqs1,
318         mem_mem_dqs_n                                   => mem_dqs_n1,
319         mem_mem_dq                                      => mem_dq1,
320         mem_mem_dm                                      => mem_dm1,
321         oct_oct_rzqin                                   => oct_rzqin1,
322         local_reset_req_local_reset_req                 => global_resetn,
323 
324         --! NIOS Status exports
325         status_input_external_connection_export         => status_wire_in,
326         status_output_external_connection_export        => status_wire_out,
327 
328         --! Master read conduits
329         master_template_0_control_fixed_location        => '0',                                      -- asserted if need a
          ↪  moving start address
330         master_template_0_control_read_base             => std_logic_vector(to_unsigned(0,32)),          -- start read address
331         master_template_0_control_read_length           => std_logic_vector(to_unsigned(644245504,32)), -- number of bytes to
          ↪  read
```

```
332         master_template_0_control_go                => go,                                  -- asserted to start
            ↪  read
333         master_template_0_control_done              => control_done_0,                      -- asserted when all
            ↪  bytes are read
334         master_template_0_user_buffer_output_data   => obtained_data_0,                     -- data obtained from
            ↪  the read
335         master_template_0_user_data_available       => data_available_0,                    -- asserted when a valid
            ↪  read has returned
336         master_template_0_user_read_buffer          => read_all,                            -- assered when the last
            ↪  buffer was read
337         master_template_1_control_fixed_location    => '0',
338         master_template_1_control_read_base         => std_logic_vector(to_unsigned(64,32)),
339         master_template_1_control_read_length       => std_logic_vector(to_unsigned(644245504,32)),
340         master_template_1_control_go                => go,
341         master_template_1_control_done              => control_done_1,
342         master_template_1_user_buffer_output_data   => obtained_data_1,
343         master_template_1_user_data_available       => data_available_1,
344         master_template_1_user_read_buffer          => read_all,
345         master_template_2_control_fixed_location    => '0',
346         master_template_2_control_read_base         => std_logic_vector(to_unsigned(128,32)),
347         master_template_2_control_read_length       => std_logic_vector(to_unsigned(644245504,32)),
348         master_template_2_control_go                => go,
349         master_template_2_control_done              => control_done_2,
350         master_template_2_user_buffer_output_data   => obtained_data_2,
351         master_template_2_user_data_available       => data_available_2,
352         master_template_2_user_read_buffer          => read_all
353     );
354
355     -- Generation of an 80MHz clock from the MGT clock
356     proc_gen_80m_clk : process (clk_320)
357     begin
358       if rising_edge(clk_320) then
359         if (unsigned(clk_counter) < 2) then
360           clk_counter <= std_logic_vector(unsigned(clk_counter) + '1');
361           clk_80m     <= '1';
362         else
363           clk_counter <= std_logic_vector(unsigned(clk_counter) + '1');
364           clk_80m     <= '0';
365         end if;
366       end if;
367     end process;
368
369     -- Generation of an 40MHz LHC BC signal
370     proc_gen_40m_enable : process (clk_80m)
371     begin
372       if rising_edge(clk_80m) then
373         enable_40m <= not enable_40m;
374       end if;
375     end process;
376
377     falling_edge_detector : entity misc.hilo_detect
378       port map (
379         clk     => clk_80m,
380         sig_in  => (status_wire_out or not(control_done_0 and control_done_1 and control_done_2)),
381         sig_out => go
382       );
383
384     -- Transferring the first master reads to a buffer
385     proc_transfer_data_0 : process (clk_80m)
386     begin
387       if rising_edge(clk_80m) then
388         if (data_available_0) then
389           master_buffer_0 <= obtained_data_0;
390         else
391           master_buffer_0 <= master_buffer_0;
392         end if;
393       end if;
394     end process;
395
396     -- Transferring the second master reads to a buffer
397     proc_transfer_data_1 : process (clk_80m)
```

```
398     begin
399       if rising_edge(clk_80m) then
400         if (data_available_1) then
401           master_buffer_1 <= obtained_data_1;
402         else
403           master_buffer_1 <= master_buffer_1;
404         end if;
405       end if;
406     end process;
407
408     -- Transferring the third master reads to a buffer
409     proc_transfer_data_2 : process (clk_80m)
410     begin
411       if rising_edge(clk_80m) then
412         if (data_available_2) then
413           master_buffer_2 <= obtained_data_2;
414         else
415           master_buffer_2 <= master_buffer_2;
416         end if;
417       end if;
418     end process;
419
420     -- Assert status signals when all three reads have returned
421     proc_read_all_buffers : process (clk_80m)
422     begin
423       if rising_edge(clk_80m) then
424         if (data_available_0 and data_available_1 and data_available_2 and not(first_read) and not(fifo_1_full)) then
425           read_all   <= '1';
426           first_read <= '1';
427         else
428           read_all   <= '0';
429           first_read <= '0';
430         end if;
431       end if;
432     end process;
433
434     --! Concatanate all buffers when all reads have returned
435     proc_concatenate_reads : process (clk_80m)
436     begin
437       if rising_edge(clk_80m) then
438         if (read_all) then
439           full_read <= master_buffer_2 & master_buffer_1 & master_buffer_0;
440         else
441           full_read <= full_read;
442         end if;
443       end if;
444     end process;
445
446     -- Instantiating a DC FIFO with a 1536-bit input and a 192-bit output
447     fifo_ram : fifo_dualwidth
448       port map (
449         data        => full_read,
450         wrreq       => read_all,
451         wrclk       => clk_80m,
452         rdreq       => enable_40m,
453         rdclk       => clk_80m,
454         q           => payload_nocheck,
455         rdempty     => fifo_1_empty,
456         wrfull      => fifo_1_full
457       );
458
459     --! Propagte signal delay
460     proc_fifo_empty_delay : process (clk_80m)
461     begin
462       if rising_edge(clk_80m) then
463         if (fifo_1_empty) then
464           fifo_1_empty_delay <= '1';
465         else
466           fifo_1_empty_delay <= '0';
467         end if;
468       end if;
```

```
469     end process;
470
471     --! Add checksum to the 192-bit buffer
472     proc_add_checksum : process (clk_80m)
473     begin
474       if rising_edge(clk_80m) then
475         if (enable_40m and not(fifo_1_empty_delay)) then
476           payload_checksum  <= check_sum & payload_nocheck;
477           transfer_complete <= '0';
478         elsif (fifo_1_empty_delay) then
479           payload_checksum  <= payload_checksum;
480           transfer_complete <= '1';
481         else
482           payload_checksum  <= payload_checksum;
483           transfer_complete <= '0';
484         end if;
485       end if;
486     end process;
487
488     --! Instantiating a DC FIFO with a clock-crossing input/output domain.
489     fifo_clk_cross : clk_crossing
490       port map (
491         data        => ((5 downto 0 => '0') & payload_checksum),
492         wrreq       => enable_40m and not(transfer_complete),
493         wrclk       => clk_80m,
494         rdreq       => enable,
495         rdclk       => clk_320,
496         q           => payload_320m,
497         rdempty     => fifo_2_empty
498       );
499
500     --! Creating a slow control clock and respective resets asserted during boot
501     slow_clk_counter : entity misc.counting
502     generic map (
503         COUNTER_MAX_VALUE => 512,
504         CYCLIC            => true
505       )
506       port map (
507         clk        => clk_80m,
508         rst        => not cpu_resetn1,
509         en         => '1',
510         cycle_done => slow_clk_count
511       );
512
513     proc_gen_slow_clk : process (clk_80m)
514     begin
515       if rising_edge(clk_80m) then
516         if (slow_clk_count =  '1') then
517           slow_clk <= not slow_clk;
518         else
519           slow_clk <= slow_clk;
520         end if;
521       end if;
522     end process;
523
524     reset_counter1 : entity misc.counting
525     generic map (
526         COUNTER_MAX_VALUE => 32766,
527         CYCLIC            => false
528       )
529       port map (
530         clk        => slow_clk,
531         rst        => not cpu_resetn1,
532         en         => '1',
533         cycle_done => reset1_counter
534       );
535
536     proc_gen_rst1 : process (slow_clk)
537     begin
538       if rising_edge(slow_clk) then
539         if (cpu_resetn1 = '0') then
```

```
540           enet_resetn <= '0';
541         elsif (reset1_counter =  '1') then
542           enet_resetn <= '1';
543         else
544           enet_resetn <= enet_resetn;
545         end if;
546       end if;
547     end process;
548
549     reset_counter2 : entity misc.counting
550     generic map (
551         COUNTER_MAX_VALUE => 65534,
552         CYCLIC            => false
553       )
554       port map (
555         clk        => slow_clk,
556         rst        => not cpu_resetn1,
557         en         => '1',
558         cycle_done => reset2_counter
559       );
560
561     proc_gen_rst2 : process (slow_clk)
562     begin
563       if rising_edge(slow_clk) then
564         if (cpu_resetn1 = '0') then
565           global_resetn <= '0';
566         elsif (reset2_counter =  '1') then
567           global_resetn <= '1';
568         else
569           global_resetn <= global_resetn;
570         end if;
571       end if;
572     end process;
573
574   end behavioral;
```

Appendix B

LASP modification: Checksum Code

```
1  ------------------------------------------------------------------------------
2  -- EMACS settings: -*- tab-width: 2; indent-tabs-mode: nil -*-
3  -- vim: tabstop=2:shiftwidth=2:expandtab
4  -- kate: tab-width 2; replace-tabs on; indent-width 2;
5  ------------------------------------------------------------------------------
6  --! @file
7  --! @author Maheyer J, Shroff <maheyer.jamshed.shroff@cern.ch>
8  --! @brief Simple entity that checks the accuracy of the payload generated
9  --! from the Injector.
10 ------------------------------------------------------------------------------
11 --! @details Instantites an entity that checks the accuracy of the payload.
12 --! from the injector. The injector's payload can be configured to have a
13 --! a checksum in bits (223 downto 192). This entity verifies the data bits
14 --! (191 downto 0) against the cheksum bits. The injector prepares the payload
15 --! such that the XOR of all the 224 user bits in 32-bit bunches is zero.
16 --!
17 --! @section inst_checkrec Instantiation template:
18 --!
19 --! @code{.vhdl}
20 --! [inst_name]: entity injector.checksum_payload
21 --!  port map (
22 --!    -- 320 MHz clock
23 --!    clk_320                 => [in std_logic],
24 --!
25 --!    -- Receiver strobe (1/8 of clock)
26 --!    enable                  => [in std_logic],
27 --!
28 --!    -- Received downlink data
29 --!    payload                 => [in std_logic_vector(229 downto 0)],
30 --!
31 --!    -- counter keeping track of inaccurate transmissions
32 --!    error_count             => [out std_logic_vector(55 downto 0)]
33 --!  );
34 --! @endcode
35 ------------------------------------------------------------------------------
36 --! @cond
37 library IEEE;
38 context IEEE.IEEE_STD_CONTEXT;
39
40 library PoC;
41   use PoC.utils.all;
42 --! @endcond
43
44 entity checksum_payload is
45   port (
46     clk_320                 : in    std_logic;                                  --! 320 MHz clock
47     enable                  : in    std_logic;                                  --! Receiver strobe (1/8 of clock)
48     payload                 : in    std_logic_vector(229 downto 0);             --! Received downlink data
```

```
49      error_count               : out   std_logic_vector(55 downto 0) := (others => '0')         --! counter incremented if error occurs
50    );
51  end checksum_payload;
52
53  --! Implementation of checksum_payload
54
55  architecture behavioral of checksum_payload is
56
57    --! result of XOR the payload in 32-bit bunches
58    signal xor_result       : std_logic_vector(31 downto 0) := (others => '0');
59
60  begin
61
62    --! Calculate the XOR of the payload broken in 32-bit bunches
63    xor_result <= (payload(31 downto 0) xor payload(63 downto 32) xor payload(95 downto 64) xor payload(127 downto 96) xor payload(159
     ↪  downto 128) xor payload(191 downto 160) xor payload(223 downto 192));
64
65    --! If the result is non zero, increment the error counter.
66    proc_increase_error : process (clk_320)
67    begin
68      if rising_edge(clk_320) then
69        if (enable) then
70          if (unsigned(payload)=0) then
71            error_count <= error_count;
72          elsif (unsigned(xor_result)/=0) then
73            error_count <= std_logic_vector((unsigned(error_count) + 1));
74          end if;
75        else
76          error_count <= error_count;
77        end if;
78      end if;
79
80    end process;
81
82  end behavioral;
```

Appendix C

Ethernet Socket Code

C.1 Socket Server (Running on FPGA)

```
/*********************************************************************
* Copyright (c) 2006 Altera Corporation, San Jose, California, USA.           *
* All rights reserved. All use of this software and documentation is          *
* subject to the License Agreement located at the end of this file below.     *
**********************************************************************
* Module - simple_socket_server.c                                            *
*                                                                            *
*********************************************************************/

/*********************************************************************
 * Simple Socket Server (SSS) example.
 *
 * This example demonstrates the use of MicroC/OS-II running on NIOS II.
 * In addition it is to serve as a good starting point for designs using
 * MicroC/OS-II and Altera NicheStack TCP/IP Stack - NIOS II Edition.
 *
 * -Known Issues
 *     None.
 *
 * Please refer to the Altera NicheStack Tutorial documentation for details on this
 * software example, as well as details on how to configure the NicheStack TCP/IP
 * networking stack and MicroC/OS-II Real-Time Operating System.
 */
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <time.h>
#include <malloc.h>
#include <unistd.h>  // usleep (unix standard?)

/* MicroC/OS-II definitions */
#include "includes.h"
#include "io.h"

/* Simple Socket Server definitions */
#include "simple_socket_server.h"
#include "alt_error_handler.h"

/* Nichestack definitions */
#include "ipport.h"
#include "tcpport.h"

/* safe string library */
#include "safe_lib.h"
```

```
45 #include <sys/time.h>
46 #include <fcntl.h>
47
48
49 /*
50  * Global handles (pointers) to our MicroC/OS-II resources. All of resources
51  * beginning with "SSS" are declared and created in this file.
52  */
53
54 /*
55  * This SSSLEDCommandQ MicroC/OS-II message queue will be used to communicate
56  * between the simple socket server task and Nios Development Board LED control
57  * tasks.
58  *
59  * Handle to our MicroC/OS-II Command Queue and variable definitions related to
60  * the Q for sending commands received on the TCP-IP socket from the
61  * SSSSimpleSocketServerTask to the LEDManagementTask.
62  */
63 OS_EVENT   *SSSLEDCommandQ;
64 #define SSS_LED_COMMAND_Q_SIZE   30   /* Message capacity of SSSLEDCommandQ */
65 void *SSSLEDCommandQTbl[SSS_LED_COMMAND_Q_SIZE]; /*Storage for SSSLEDCommandQ*/
66
67
68 /*
69  * Handle to our MicroC/OS-II LED Event Flag.  Each flag corresponds to one of
70  * the LEDs on the Nios Development board, D0 - D7.
71  */
72 OS_FLAG_GRP *SSSLEDEventFlag;
73
74 /*
75  * Handle to our MicroC/OS-II LED Lightshow Semaphore. The semaphore is checked
76  * by the LED7SegLightshowTask each time it updates 7 segment LED displays,
77  * U8 and U9.  The LEDManagementTask grabs the semaphore away from the lightshow task to
78  * toggle the lightshow off, and gives up the semaphore to turn the lightshow
79  * back on.  The LEDManagementTask does this in response to the CMD_LEDS_LIGHTSHOW
80  * command sent from the SSSSimpleSocketServerTask when the user sends a toggle
81  * lightshow command over the TCPIP socket.
82  */
83 OS_EVENT *SSSLEDLightshowSem;
84
85 /* Definition of Task Stacks for tasks not invoked by TK_NEWTASK
86  * (do not use NicheStack)
87  */
88
89 OS_STK     LEDManagementTaskStk[TASK_STACKSIZE];
90 OS_STK     LED7SegLightshowTaskStk[TASK_STACKSIZE];
91
92 /*
93  * Create our MicroC/OS-II resources. All of the resources beginning with
94  * "SSS" are declared in this file, and created in this function.
95  */
96 void SSSCreateOSDataStructs(void)
97 {
98   INT8U error_code;
99
100   /*
101   * Create the resource for our MicroC/OS-II Queue for sending commands
102   * received on the TCP/IP socket from the SSSSimpleSocketServerTask()
103   * to the LEDManagementTask().
104   */
105   SSSLEDCommandQ = OSQCreate(&SSSLEDCommandQTbl[0], SSS_LED_COMMAND_Q_SIZE);
106   if (!SSSLEDCommandQ)
107   {
108      alt_uCOSIIErrorHandler(EXPANDED_DIAGNOSIS_CODE,
109      "Failed to create SSSLEDCommandQ.\n");
110   }
111
112  /* Create our MicroC/OS-II LED Lightshow Semaphore.  The semaphore is checked
113   * by the SSSLEDLightshowTask each time it updates 7 segment LED displays,
114   * U8 and U9.  The LEDTask grabs the semaphore away from the lightshow task to
115   * toggle the lightshow off, and gives up the semaphore to turn the lightshow
```

```
116   * back on.  The LEDTask does this in response to the CMD_LEDS_LIGHTSHOW
117   * command sent from the SSSSimpleSocketServerTask when the user sends the
118   * toggle lightshow command over the TCPIP socket.
119   */
120   SSSLEDLightshowSem = OSSemCreate(1);
121   if (!SSSLEDLightshowSem)
122   {
123      alt_uCOSIIErrorHandler(EXPANDED_DIAGNOSIS_CODE,
124                             "Failed to create SSSLEDLightshowSem.\n");
125   }
126
127  /*
128   * Create our MicroC/OS-II LED Event Flag.  Each flag corresponds to one of
129   * the LEDs on the Nios Development board, D0 - D7.
130   */
131   SSSLEDEventFlag = OSFlagCreate(0, &error_code);
132   if (!SSSLEDEventFlag)
133   {
134      alt_uCOSIIErrorHandler(error_code, 0);
135   }
136 }
137
138 /* This function creates tasks used in this example which do not use sockets.
139  * Tasks which use Interniche sockets must be created with TK_NEWTASK.
140  */
141
142 void SSSCreateTasks(void)
143 {
144    INT8U error_code;
145
146
147    alt_uCOSIIErrorHandler(error_code, 0);
148
149    alt_uCOSIIErrorHandler(error_code, 0);
150
151 }
152
153
154 /*
155  * sss_reset_connection()
156  *
157  * This routine will, when called, reset our SSSConn struct's members
158  * to a reliable initial state. Note that we set our socket (FD) number to
159  * -1 to easily determine whether the connection is in a "reset, ready to go"
160  * state.
161  */
162 void sss_reset_connection(SSSConn* conn)
163 {
164   memset(conn, 0, sizeof(SSSConn));
165
166   conn->fd = -1;
167   conn->state = READY;
168   conn->rx_wr_pos = conn->rx_buffer;
169   conn->rx_rd_pos = conn->rx_buffer;
170   return;
171 }
172
173 /*
174  * sss_send_menu()
175  *
176  * This routine will transmit the menu out to the telent client.
177  */
178 void sss_send_menu(SSSConn* conn)
179 {
180   char  tx_buf[SSS_TX_BUF_SIZE];
181   char *tx_wr_pos = tx_buf;
182
183
184   tx_wr_pos += sprintf(tx_wr_pos,"================================\n\r");
185   tx_wr_pos += sprintf(tx_wr_pos,"LASP Injector Socket Menu\n\r");
186   tx_wr_pos += sprintf(tx_wr_pos,"================================\n\r");
```

```
187   tx_wr_pos += sprintf(tx_wr_pos,"ACQUIRE      - Obtain data from Workstation \n\r");
188   tx_wr_pos += sprintf(tx_wr_pos,"TRANSFER     - Transfer data to H-tile\n\r");
189   tx_wr_pos += sprintf(tx_wr_pos,"MENU         - display this menu \n\r");
190   tx_wr_pos += sprintf(tx_wr_pos,"QUIT         - quit \n\r");
191   tx_wr_pos += sprintf(tx_wr_pos,"==========================================\n\r");
192   tx_wr_pos += sprintf(tx_wr_pos,"Enter your choice & press return:\n\r");
193
194   send(conn->fd, tx_buf, tx_wr_pos - tx_buf, 0);
195
196     return;
197 }
198
199 /*
200  * sss_handle_accept()
201  *
202  * This routine is called when ever our listening socket has an incoming
203  * connection request. Since this example has only data transfer socket,
204  * we just look at it to see whether its in use... if so, we accept the
205  * connection request and call the telent_send_menu() routine to transmit
206  * instructions to the user. Otherwise, the connection is already in use;
207  * reject the incoming request by immediately closing the new socket.
208  *
209  * We'll also print out the client's IP address.
210  */
211 void sss_handle_accept(int listen_socket, SSSConn* conn)
212 {
213   int                 socket, len;
214   struct sockaddr_in  incoming_addr;
215
216   len = sizeof(incoming_addr);
217
218   if ((conn)->fd == -1)
219   {
220      if((socket=accept(listen_socket,(struct sockaddr*)&incoming_addr,&len))<0)
221      {
222          alt_NetworkErrorHandler(EXPANDED_DIAGNOSIS_CODE,
223                                  "[sss_handle_accept] accept failed");
224      }
225      else
226      {
227         (conn)->fd = socket;
228         sss_send_menu(conn);
229         printf("[sss_handle_accept] accepted connection request from %s\n",
230                inet_ntoa(incoming_addr.sin_addr));
231      }
232   }
233   else
234   {
235     printf("[sss_handle_accept] rejected connection request from %s\n",
236            inet_ntoa(incoming_addr.sin_addr));
237   }
238
239   return;
240 }
241
242 /*
243  * sss_exec_command()
244  *
245  * This routine is called whenever we have new, valid receive data from our
246  * sss connection. It will parse through the data simply looking for valid
247  * commands to the sss server.
248  *
249  * Incoming commands to talk to the board LEDs are handled by sending the
250  * MicroC/OS-II SSSLedCommandQ a pointer to the value we received.
251  *
252  * If the user wishes to quit, we set the "close" member of our SSSConn
253  * struct, which will be looked at back in sss_handle_receive() when it
254  * comes time to see whether to close the connection or not.
255  */
256 void sss_exec_command(SSSConn* conn)
257 {
```

```
258    int bytes_to_process = conn->rx_wr_pos - conn->rx_rd_pos;
259
260    char  tx_buf[SSS_TX_BUF_SIZE];
261    char *tx_wr_pos = tx_buf;
262
263    char text_buf[SSS_TX_BUF_SIZE];
264    char *text = text_buf;
265    INT8U error_code;
266
267    char  tx_buf_data[SSS_TX_BUF_SIZE];
268    char *tx_wr_pos_data = tx_buf_data;
269    /*
270     * "SSSCommand" is declared static so that the data will reside
271     * in the BSS segment. This is done because a pointer to the data in
272     * SSSCommand
273     * will be passed via SSSLedCommandQ to the LEDManagementTask.
274     * Therefore SSSCommand cannot be placed on the stack of the
275     * SSSSimpleSocketServerTask, since the LEDManagementTask does not
276     * have access to the stack of the SSSSimpleSocketServerTask.
277     */
278    static INT32U SSSCommand;
279
280    SSSCommand = CMD_LEDS_BIT_0_TOGGLE;
281
282    while(bytes_to_process--)
283    {
284     SSSCommand = tolower(*(conn->rx_rd_pos++));
285     text += sprintf(text,"%c",(char)SSSCommand);
286    }
287
288    if(strstr(text_buf, "acquire") != NULL)
289    {
290      typedef alt_u16 my_data;
291      int data_size;
292      recv(conn->fd, &data_size, sizeof(data_size),0);
293
294      char *pBuffer = malloc(data_size*sizeof(my_data));
295      printf("Number of bytes to receive %d\n",(data_size*sizeof(data_size)));
296
297      int expected_size = sizeof(my_data) * data_size;
298      int received_bytes = 0, received_status = 1;
299      while(received_bytes < expected_size && received_status > 0)
300      {
301        received_status = recv(conn->fd, pBuffer, expected_size - received_status,0);
302        memcpy(EMIF_S10_0_BASE+received_bytes,pBuffer,received_status);
303        pBuffer += received_status;
304        received_bytes += received_status;
305      }
306      alt_dcache_flush_all();
307      free(pBuffer);
308
309   else if ( strstr(text_buf, "quit") != NULL)
310   {
311      tx_wr_pos += sprintf(tx_wr_pos,"Terminating connection.\n\n\r");
312      conn->close = 1;
313   }
314   else if ( strstr(text_buf, "menu") != NULL)
315   {
316      sss_send_menu(conn);
317   }
318
319   else if ( strstr(text_buf, "transfer") != NULL)
320     {
321     IOWR(STATUS_OUTPUT_BASE,0,0x1);
322     IOWR(STATUS_OUTPUT_BASE,0,0x0); // Pulse status output to signal completion of transfer
323     }
324
325 send(conn->fd, tx_buf, tx_wr_pos - tx_buf, 0);
326
327 return;
328 }
```

```
329
330 /*
331  * sss_handle_receive()
332  *
333  * This routine is called whenever there is a sss connection established and
334  * the socket assocaited with that connection has incoming data. We will first
335  * look for a newline "\n" character to see if the user has entered something
336  * and pressed 'return'. If there is no newline in the buffer, we'll attempt
337  * to receive data from the listening socket until there is.
338  *
339  * The connection will remain open until the user enters "Q\n" or "q\n", as
340  * deterimined by repeatedly calling recv(), and once a newline is found,
341  * calling sss_exec_command(), which will determine whether the quit
342  * command was received.
343  *
344  * Finally, each time we receive data we must manage our receive-side buffer.
345  * New data is received from the sss socket onto the head of the buffer,
346  * and popped off from the beginning of the buffer with the
347  * sss_exec_command() routine. Aside from these, we must move incoming
348  * (un-processed) data to buffer start as appropriate and keep track of
349  * associated pointers.
350  */
351 void sss_handle_receive(SSSConn* conn)
352 {
353   int data_used = 0, rx_code = 0;
354   char *lf_addr;
355
356   conn->rx_rd_pos = conn->rx_buffer;
357   conn->rx_wr_pos = conn->rx_buffer;
358
359   printf("[sss_handle_receive] processing RX data\n");
360
361   while(conn->state != CLOSE)
362   {
363     /* Find the Carriage return which marks the end of the header */
364     lf_addr = strchr((const char*)conn->rx_buffer, '\n');
365
366     if(lf_addr)
367     {
368       /* go off and do whatever the user wanted us to do */
369       sss_exec_command(conn);
370     }
371     /* No newline received? Then ask the socket for data */
372     else
373     {
374       rx_code = recv(conn->fd, (char*)conn->rx_wr_pos,
375         SSS_RX_BUF_SIZE - (conn->rx_wr_pos - conn->rx_buffer) -1, 0);
376
377      if(rx_code > 0)
378       {
379         conn->rx_wr_pos += rx_code;
380
381         /* Zero terminate so we can use string functions */
382         *(conn->rx_wr_pos+1) = 0;
383       }
384     }
385
386     /*
387      * When the quit command is received, update our connection state so that
388      * we can exit the while() loop and close the connection
389      */
390     conn->state = conn->close ? CLOSE : READY;
391
392     /* Manage buffer */
393     data_used = conn->rx_rd_pos - conn->rx_buffer;
394     memmove_s(conn->rx_buffer,
395        conn->rx_wr_pos - conn->rx_rd_pos,
396        conn->rx_rd_pos,
397        conn->rx_wr_pos - conn->rx_rd_pos);
398     conn->rx_rd_pos = conn->rx_buffer;
399     conn->rx_wr_pos -= data_used;
```

```
400     memset(conn->rx_wr_pos, 0, data_used);
401   }
402
403   printf("[sss_handle_receive] closing connection\n");
404   close(conn->fd);
405   sss_reset_connection(conn);
406
407   return;
408 }
409
410 /*
411  * SSSSimpleSocketServerTask()
412  *
413  * This MicroC/OS-II thread spins forever after first establishing a listening
414  * socket for our sss connection, binding it, and listening. Once setup,
415  * it perpetually waits for incoming data to either the listening socket, or
416  * (if a connection is active), the sss data socket. When data arrives,
417  * the approrpriate routine is called to either accept/reject a connection
418  * request, or process incoming data.
419  */
420 void SSSSimpleSocketServerTask()
421 {
422   int fd_listen, max_socket;
423   struct sockaddr_in addr;
424   static SSSConn conn;
425   fd_set readfds;
426
427   /*
428    * Sockets primer...
429    * The socket() call creates an endpoint for TCP of UDP communication. It
430    * returns a descriptor (similar to a file descriptor) that we call fd_listen,
431    * or, "the socket we're listening on for connection requests" in our sss
432    * server example.
433    *
434    * Traditionally, in the Sockets API, PF_INET and AF_INET is used for the
435    * protocol and address families respectively. However, there is usually only
436    * 1 address per protocol family. Thus PF_INET and AF_INET can be interchanged.
437    * In the case of NicheStack, only the use of AF_INET is supported.
438    * PF_INET is not supported in NicheStack.
439    */
440   if ((fd_listen = socket(AF_INET, SOCK_STREAM, 0)) < 0)
441   {
442     alt_NetworkErrorHandler(EXPANDED_DIAGNOSIS_CODE,"[sss_task] Socket creation failed");
443   }
444   /*
445    * Sockets primer, continued...
446    * Calling bind() associates a socket created with socket() to a particular IP
447    * port and incoming address. In this case we're binding to SSS_PORT and to
448    * INADDR_ANY address (allowing anyone to connect to us. Bind may fail for
449    * various reasons, but the most common is that some other socket is bound to
450    * the port we're requesting.
451    */
452   addr.sin_family = AF_INET;
453   addr.sin_port = htons(SSS_PORT);
454   addr.sin_addr.s_addr = INADDR_ANY;
455
456   if ((bind(fd_listen,(struct sockaddr *)&addr,sizeof(addr))) < 0)
457   {
458     alt_NetworkErrorHandler(EXPANDED_DIAGNOSIS_CODE,"[sss_task] Bind failed");
459   }
460
461   /*
462    * Sockets primer, continued...
463    * The listen socket is a socket which is waiting for incoming connections.
464    * This call to listen will block (i.e. not return) until someone tries to
465    * connect to this port.
466    */
467   if ((listen(fd_listen,1)) < 0)
468   {
469     alt_NetworkErrorHandler(EXPANDED_DIAGNOSIS_CODE,"[sss_task] Listen failed");
470   }
```

```
471
472   /* At this point we have successfully created a socket which is listening
473    * on SSS_PORT for connection requests from any remote address.
474    */
475   sss_reset_connection(&conn);
476   printf("[sss_task] Simple Socket Server listening on port %d\n", SSS_PORT);
477
478   while(1)
479   {
480     /*
481      * For those not familiar with sockets programming...
482      * The select() call below basically tells the TCPIP stack to return
483      * from this call when any of the events I have expressed an interest
484      * in happen (it blocks until our call to select() is satisfied).
485      *
486      * In the call below we're only interested in either someone trying to
487      * connect to us, or data being available to read on a socket, both of
488      * these are a read event as far as select is called.
489      *
490      * The sockets we're interested in are passed in in the readfds
491      * parameter, the format of the readfds is implementation dependant
492      * Hence there are standard MACROs for setting/reading the values:
493      *
494      *   FD_ZERO  - Zero's out the sockets we're interested in
495      *   FD_SET   - Adds a socket to those we're interested in
496      *   FD_ISSET - Tests whether the chosen socket is set
497      */
498     FD_ZERO(&readfds);
499     FD_SET(fd_listen, &readfds);
500     max_socket = fd_listen+1;
501
502     if (conn.fd != -1)
503     {
504       FD_SET(conn.fd, &readfds);
505       if (max_socket <= conn.fd)
506       {
507         max_socket = conn.fd+1;
508       }
509     }
510
511     select(max_socket, &readfds, NULL, NULL, NULL);
512
513     /*
514      * If fd_listen (the listening socket we originally created in this thread
515      * is "set" in readfs, then we have an incoming connection request. We'll
516      * call a routine to explicitly accept or deny the incoming connection
517      * request (in this example, we accept a single connection and reject any
518      * others that come in while the connection is open).
519      */
520     if (FD_ISSET(fd_listen, &readfds))
521     {
522       sss_handle_accept(fd_listen, &conn);
523     }
524     /*
525      * If sss_handle_accept() accepts the connection, it creates *another*
526      * socket for sending/receiving data over sss. Note that this socket is
527      * independant of the listening socket we created above. This socket's
528      * descriptor is stored in conn.fd. If conn.fs is set in readfs... we have
529      * incoming data for our sss server, and we call our receiver routine
530      * to process it.
531      */
532
533     else
534     {
535       if ((conn.fd != -1) && FD_ISSET(conn.fd, &readfds))
536       {
537         sss_handle_receive(&conn);
538       }
539     }
540   } /* while(1) */
541 }
```

```
542
543
544
545 /*****************************************************************************
546 *                                                                           *
547 * License Agreement                                                         *
548 *                                                                           *
549 * Copyright (c) 2009 Altera Corporation, San Jose, California, USA.         *
550 * All rights reserved.                                                      *
551 *                                                                           *
552 * Permission is hereby granted, free of charge, to any person obtaining a   *
553 * copy of this software and associated documentation files (the "Software"), *
554 * to deal in the Software without restriction, including without limitation *
555 * the rights to use, copy, modify, merge, publish, distribute, sublicense,  *
556 * and/or sell copies of the Software, and to permit persons to whom the     *
557 * Software is furnished to do so, subject to the following conditions:      *
558 *                                                                           *
559 * The above copyright notice and this permission notice shall be included in *
560 * all copies or substantial portions of the Software.                       *
561 *                                                                           *
562 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR *
563 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,  *
564 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE *
565 * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER    *
566 * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING   *
567 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER       *
568 * DEALINGS IN THE SOFTWARE.                                                 *
569 *                                                                           *
570 * This agreement shall be governed in all respects by the laws of the State *
571 * of California and by the laws of the United States of America.            *
572 * Altera does not recommend, suggest or require that this reference design  *
573 * file be used in conjunction or combination with any other product.        *
574 *****************************************************************************/
```

C.2 Socket Client (bulk_client_send.C running on workstation)

```
/*****************************************************************************
 * Socket Client .
 *
 * This piece of code runs on the workstation.
 * It is developed using socket API to attempt a connection of a known IP
 * address and port number.
 * Once the connection is made, the program communicates necessary
 * preliminary information with the Socket server on the FPGA after which it
 * transfers 1.8GB of an increasing 16-bit counter.
 * The socket connection is then closed.
 *
 * USAGE:
 *     ./<executable from bulk_client_send.C> <IP_ADDRESS> <PORT NUMBER>
 *****************************************************************************/

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdlib.h>
#include <strings.h>
#include <string.h>
#include <time.h>
#include <arpa/inet.h>
#include <unistd.h>


#define DEPTH 966367641          // NUMBER OF 16-BIT ADC VALUES TO BE SENT

void error(char * msg) {
  perror(msg);
  exit(0);
}

int main(int argc, char * argv[]) {

  int sock, portno, n;
  struct sockaddr_in serv_addr;
  struct hostent * server;

  char buffer[256];
  if (argc < 3) {
    fprintf(stderr, "usage %s hostname port\n", argv[0]);
    exit(0);
  }

  portno = atoi(argv[2]);
  sock = socket(AF_INET, SOCK_STREAM, 0);
  if (sock < 0)
    error("ERROR opening socket");
  server = gethostbyname(argv[1]);
  if (server == NULL) {
    fprintf(stderr, "ERROR, no such host\n");
    exit(0);
  }
  bzero((char * ) & serv_addr, sizeof(serv_addr));
  serv_addr.sin_family = AF_INET;
  bcopy((char * ) server -> h_addr,
    (char * ) & serv_addr.sin_addr.s_addr,
    server -> h_length);
  serv_addr.sin_port = htons(portno);
  if (connect(sock, (struct sockaddr * ) & serv_addr, sizeof(serv_addr)) < 0)
    error("ERROR connecting");

```

```
66    // IF PROGRAM REACHES THIS POINT, SOCKET CONNECTION IS SUCCESSFUL.
67    char menu_buffer[5024] = {0};
68    read(sock, & menu_buffer, 1024);
69
70    char acquire_text[1024] = "acquire\n";
71    send(sock, & acquire_text, sizeof(acquire_text), 0);
72    usleep(1000000);
73
74    int data_size = DEPTH;
75    send(sock, & data_size, sizeof(data_size), 0);
76    usleep(1000000);
77
78    static short send_buff[DEPTH];
79    float rec_buff;
80
81    for (int i = 0; i < data_size; i++) {
82      send_buff[i] = (short) i + 1;
83    }
84    int sent_bytes = send(sock, & send_buff, sizeof(send_buff), 0);
85
86    printf("Bytes sent: %d\n", sent_bytes);
87
88    char closed;
89    int closed_check = 1;
90    char transfer_text[1024] = "transfer\n";
91    usleep(500000);
92    send(sock, & transfer_text, sizeof(transfer_text), 0);
93
94    usleep(500000);
95    char quit_text[1024] = "quit\n";
96    send(sock, & quit_text, sizeof(quit_text), 0);
97
98    return 0;
99  }
```

Bibliography

[1] Michael E. Peskin and Daniel V. Schroeder. *An Introduction to Quantum Field Theory*. Addison-Wesley, Reading, USA, 1995.

[2] Matthew D. Schwartz. *Quantum Field Theory and the Standard Model*. Cambridge University Press, 3 2014.

[3] D.H. Perkins. *Introduction to High Energy Physics*. 1 1982.

[4] I.J.R. Aitchison and A.J.G. Hey. *Gauge Theories in Particle physics: A practical introduction. Vol. 2: Non-Abelian gauge theories: QCD and the electroweak theory*. CRC Press, 2012.

[5] David Griffiths. *Introduction to Elementary Particles*. 2008.

[6] Georges Aad et al. Observation of a new particle in the search for the Standard Model Higgs boson with the ATLAS detector at the LHC. Observation of a new particle in the search for the Standard Model Higgs boson with the ATLAS detector at the LHC. *Phys. Lett. B*, 716(arXiv:1207.7214. CERN-PH-EP-2012-218):1–29. 29 p, Aug 2012.

[7] Steven Weinberg. Non-abelian gauge theories of the strong interactions. *Phys. Rev. Lett.*, 31:494–497, Aug 1973.

[8] M. Tanabashi et al. Review of Particle Physics. *Phys. Rev. D*, 98:030001, Aug 2018.

[9] Eleonora Rossi. Measurement of Higgs-boson self-coupling with single-Higgs and double-Higgs production channels, 2019.

[10] S.L. Glashow. Partial Symmetries of Weak Interactions. *Nucl. Phys.*, 22:579–588, 1961.

[11] Steven Weinberg. A model of leptons. *Phys. Rev. Lett.*, 19:1264–1266, Nov 1967.

[12] F. Englert and R. Brout. Broken symmetry and the mass of gauge vector mesons. *Phys. Rev. Lett.*, 13:321–323, Aug 1964.

[13] Peter W. Higgs. Broken symmetries and the masses of gauge bosons. *Phys. Rev. Lett.*, 13:508–509, Oct 1964.

[14] Peter W. Higgs. Spontaneous symmetry breakdown without massless bosons. *Phys. Rev.*, 145:1156–1163, May 1966.

[15] Verena Maria Walbrecht. Measurement of the Higgs Boson Coupling Structure in Decays to Four Leptons with the ATLAS Detector, 2020.

[16] D. de Florian et al. Handbook of LHC Higgs Cross Sections: 4. Deciphering the Nature of the Higgs Sector. 10 2016.

[17] Prospects for measuring Higgs pair production in the channel $H(\rightarrow \gamma\gamma)H(\rightarrow b\bar{b})$ using the ATLAS detector at the HL-LHC. Technical Report ATL-PHYS-PUB-2014-019, CERN, Geneva, Oct 2014.

[18] Florian Eble. Prospective study for the development of an analysis to measure the tri-linear Higgs coupling using the Matrix Element Method with the ATLAS experiment at the HL-LHC., 2020. Presented 2020.

[19] Esma Mobs. The CERN accelerator complex - 2019. Complexe des accélérateurs du CERN - 2019. Jul 2019. General Photo.

[20] Lyndon Evans and Philip Bryant. LHC machine. *Journal of Instrumentation*, 3(08):S08001–S08001, Aug 2008.

[21] A. Airapetian et al. *ATLAS detector and physics performance: Technical Design Report, 1.* Technical Design Report ATLAS. CERN, Geneva, 1999.

[22] A. Airapetian et al. *ATLAS detector and physics performance: Technical Design Report, 2.* Technical Design Report ATLAS. CERN, Geneva, 1999.

[23] The ATLAS Collaboration. The ATLAS experiment at the CERN large hadron collider. *Journal of Instrumentation*, 3(08):S08003–S08003, Aug 2008.

[24] ATLAS Collaboration. Technical Design Report for the Phase-II Upgrade of the ATLAS LAr Calorimeter. Technical Report CERN-LHCC-2017-018. ATLAS-TDR-027, CERN, Geneva, Sep 2017.

[25] N J Buchanan et al. Design and implementation of the Front End Board for the readout of the ATLAS Liquid Argon calorimeters. *Journal of Instrumentation*, 3(03):P03004–P03004, mar 2008.

[26] M Aleksa et al. ATLAS Liquid Argon Calorimeter Phase-I Upgrade Technical Design Report. Technical Report CERN-LHCC-2013-017. ATLAS-TDR-022, Sep 2013.

[27] The VLplus project. `https://espace.cern.ch/project-Versatile-Link-Plus/SitePages/Home.aspx`. Accessed: 2020-04-20.

[28] ATLAS Collaboration. ATLAS LAr calorimeter Phase-II Upgrade: Off-Detector Electronics Project Description.

[29] ATLAS Collaboration. ATLAS LAr Calorimeter Phase-II Upgrade: LAr Signal Processor (LASP) blade System Firmware Specification.

[30] Weihao Wu. FELIX: the New Detector Interface for the ATLAS Experiment. *IEEE Trans. Nucl. Sci.*, 66(arXiv:1806.10667. 7):986–992. 7 p, Jun 2018.

[31] P.P. Chu. *Embedded SoPC Design with Nios II Processor and Verilog Examples*. Wiley, 2012.

[32] Heiko Engel. Development of a Read-Out Receiver Card for Fast Processing of Detector Data, Sep 2018. Presented 10 Jul 2019.

[33] Steffen Stärz. Development of Digital Signal Processing with FPGAs for the Readout of the ATLAS Liquid Argon Calorimeter at HL-LHC, 2010. Presented on 14 Dec 2010.

[34] Intel Stratix 10 GX FPGA Development Kit User Guide. Technical report, Intel Corporation., 2019.

[35] Intel Stratix 10 Logic Array Blocks and Adaptive Logic Modules User Guide. Technical report, Intel Corporation., 2018.

[36] Intel Stratix 10 GX/SX Product Table. Technical report, Intel Corporation., 2018.

[37] Avalon Interface Specifications. Technical report, Altera Corporation., 2010.

[38] Intel Quartus Prime Pro Edition User Guide. Technical report, Intel Corporation., 2019.

[39] J P Grohs and S Stärz. AREUS: ATLAS Readout Electronics Upgrade Simulation. Technical Report ATL-COM-LARG-2013-012, CERN, Geneva, May 2013.

[40] Steffen Stärz. Energy Reconstruction and high-speed Data Transmission with FPGAs for the Upgrade of the ATLAS Liquid Argon Calorimeter at LHC, Feb 2015. Presented 19 May 2015.

[41] Using Triple-Speed Ethernet on DE2-115 Boards. Technical report, Intel Corporation., 2017.

[42] Using the NicheStack TCP/IP Stack. Technical report, Intel Corporation., 2019.

[43] Nios II Software Developer Handbook. Technical report, Intel Corporation., 2019.

[44] DDR3L SDRAM Description - Datasheet. Technical report, Micron Technology, Inc., 2014.

[45] External Memory Interfaces Intel Stratix 10 FPGA IP User Guide. Technical report, Intel Corporation., 2020.

[46] Memory-Mapped Master Templates. Technical report, Intel Corporation., 2007.

[47] LpGBT-FPGA Documentation. Technical report, CERN, 2020.

[48] Metastability in Altera Devices. Technical report, Altera Corporation., 1999.