

**Multiple UDP ports for FaceWorks, a networked IP core  
used for debug**

**João Carlos dos Santos Fernandes**

Thesis to obtain the Master of Science Degree in

**Electrical and Computer Engineering**

**Examination Committee**

Chairperson: Prof. João Manuel Torres Caldinhas Simões Vaz

Supervisor: Prof. José João Henriques Teixeira de Sousa

Member of the Committee: Prof. Paulo Ferreira Godinho Flores

**November 2013**



# Abstract

This thesis presents an implementation of multiple User Datagram Protocol (UDP) ports for FaceWorks, an Intellectual Property (IP) core used for hardware test and debug over the UDP/IP network protocol, which is a patented and proprietary technology of the company Coreworks SA. The objective of having multiple ports is to allow multiple software processes to independently control the device under test (eg. multiple audio streams driven by multiple programs). An overview of the FaceWorks technology and its possible applications is presented, with emphasis on the Coreworks Datagram Protocol (CWDP), which was added on top of the common UDP protocol to enable test and debug functions. The FaceWorks hardware has been converted to Verilog, upgraded to support two UDP ports, and tested using the Verilator simulator and an Field Programmable Logic Array (FPGA) board. A SystemC model of the core is automatically generated and simulated by Verilator. A SystemC testbench that uses network sockets has been written to fully replicate the hardware behavior in simulation. A PC software driver and an application which connects to the two UDP ports for testing the system have been developed. This application connects to both the SystemC model or FPGA board, indistinguishably. Experiments and implementation results are reported.

# Keywords

Ethernet, UDP, CWDP, System-on-Chip, Verilator, Verification



# Resumo

Esta tese apresenta uma implementação de portas UDP múltiplas para o FaceWorks, um núcleo de Propriedade Intelectual utilizado para o teste e depuração de *hardware*. Funciona sobre o protocolo de rede UDP/Internet Protocol (IP), e é uma tecnologia patenteada e propriedade da empresa Coreworks SA. O objectivo das portas múltiplas é permitir que vários processos de *software* possam controlar de forma independente o dispositivo de teste (múltiplos fluxos de áudio gerados por vários programas). Uma descrição geral da tecnologia FaceWorks e suas possíveis aplicações é apresentada, dando ênfase ao *Coreworks Datagram Protocol (CWDP)*, que consiste numa camada adicionada sobre o protocolo UDP para realizar as funções de teste e depuração. A descrição do FaceWorks foi convertida para Verilog, actualizada para suportar duas portas UDP, e testada utilizando o simulador Verilator e uma placa de FPGA. Foi gerado automaticamente um modelo SystemC do núcleo e simulado com recurso ao Verilator. Foi descrita uma bancada de teste em SystemC que usa *sockets* de rede para replicar totalmente o comportamento do *hardware* em simulação. Foram desenvolvidos um controlador de software para PC e uma aplicação que se liga às duas portas UDP para testar o sistema. A aplicação desenvolvida interage indiferentemente com o modelo SystemC ou com a placa de FPGA. Experiências e resultados de implementação são apresentados.

## Palavras Chave

Ethernet, UDP, CWDP, Sistema-num-Chip, Verilator, Verificação



# Contents

<b>Abstract</b>	<b>i</b>
<b>Resumo</b>	<b>iii</b>
<b>Contents</b>	<b>v</b>
<b>List of Figures</b>	<b>ix</b>
<b>List of Tables</b>	<b>xi</b>
<b>List of Acronyms</b>	<b>xiii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Scope . . . . .	2
1.2 Problem . . . . .	3
1.3 Solution . . . . .	4
1.4 Objectives . . . . .	4
1.5 Outline . . . . .	4
<b>2 Background</b>	<b>7</b>
2.1 Media Independent Interface . . . . .	8
2.1.1 Data Stream . . . . .	10
2.2 Media Access Control . . . . .	10
2.3 Address Resolution Protocol . . . . .	12
2.4 Internet Protocol . . . . .	14
2.5 User Datagram Protocol . . . . .	16
2.5.1 UDP Packet Header . . . . .	16
2.5.2 Implementation . . . . .	17
2.6 Coreworks Datagram Protocol . . . . .	18
2.6.1 CWDP Packet . . . . .	18
2.6.2 Packet Types . . . . .	19
2.6.3 Implementation . . . . .	21
	<b>v</b>

## Contents

---

2.6.4	CW-Link Interface . . . . .	22
<b>3</b>	<b>Design</b>	<b>25</b>
3.1	Face-Test Application . . . . .	26
3.1.1	Basic Functions . . . . .	26
3.1.2	Test Application . . . . .	27
3.2	Hardware Implementation . . . . .	31
3.2.1	Modified Blocks . . . . .	32
3.2.2	Arbiter . . . . .	35
<b>4</b>	<b>Verification</b>	<b>37</b>
4.1	Verilator . . . . .	38
4.1.1	Verilator History . . . . .	38
4.1.2	Verilator Testbench . . . . .	39
4.2	Verification Environment . . . . .	39
4.2.1	sniffudp . . . . .	40
4.2.2	FaceWorks Controller . . . . .	41
4.2.3	process_rcv_packet . . . . .	42
4.2.4	Cyclic Redundancy Check . . . . .	42
4.3	Verilator Module Instantiation . . . . .	43
4.4	Running the Simulation Model . . . . .	46
4.4.1	Folder Structure . . . . .	46
4.4.2	Producing the Simulation Model . . . . .	46
4.4.3	Linting the Verilog Code . . . . .	47
4.4.4	Observing Wave Traces . . . . .	49
4.4.5	Coverage . . . . .	49
<b>5</b>	<b>Results</b>	<b>51</b>
5.1	FPGA Implementation Results . . . . .	52
5.2	Ethernet Switch Characterization . . . . .	53
5.3	Throughput Upper Bound Model . . . . .	55
5.4	FaceWorks Sustained Throughput . . . . .	56
<b>6</b>	<b>Conclusion</b>	<b>58</b>
6.1	Summary . . . . .	59
6.2	Future work . . . . .	60
6.2.1	Not Acknowledge . . . . .	61
6.2.2	A UDP/CWDP Cycle Accurate Simulator . . . . .	63



<b>A</b>	<b>Appendix A GTKwave</b>	<b>67</b>
----------	---------------------------	-----------



# List of Figures

2.1	FaceWorks Block Diagram . . . . .	8
2.2	Interconnection of Ethernet Physical Transceiver (PHY) and FaceWorks using MII . . . . .	9
2.3	Media Independent Interface (MII) Packet . . . . .	10
2.4	Ethernet Packet Structure with the Media Access Control (MAC) Protocol . . . . .	11
2.5	Address Resolution Protocol (ARP) Packet . . . . .	12
2.6	Address Resolution using ARP . . . . .	14
2.7	IPv4 Packet Header . . . . .	14
2.8	UDP Packet Header . . . . .	17
2.9	CWDP Encapsulation in the UDP Payload . . . . .	18
2.10	CWDP Packet . . . . .	18
2.11	CWDP SET_CORE_ACCESS Packet . . . . .	19
2.12	CWDP SET_MAC Packet . . . . .	19
2.13	CWDP CORE DATA Packet . . . . .	20
2.14	CWDP ACK Packet . . . . .	20
2.15	CWDP Layer . . . . .	21
2.16	CW-Link Interface . . . . .	22
2.17	CW-Link RX Transactions . . . . .	23
3.1	CW-Link Loopback . . . . .	26
3.2	CWDP Send/Receive Functions . . . . .	27
3.3	Send Core Access and Wait for Acknowledge . . . . .	27
3.4	Recovering from Lost Packets . . . . .	28
3.5	Send Core Data and Wait for Acknowledge . . . . .	28
3.6	Read Core Access and Send Acknowledge . . . . .	29
3.7	Facework Recovery from a Lost Packet . . . . .	30
3.8	Two-Port FaceWorks Block Diagram . . . . .	31
3.9	CWDP RX and CW-Link RX . . . . .	32
3.10	CWDP TX and CW-Link TX . . . . .	33
3.11	CWDP Layer . . . . .	34

## List of Figures

---

3.12 Arbiter Finite State Machine (FSM) State Transition Diagram . . . . .	36
4.1 Verification Environment . . . . .	39
4.2 Injected ARP Reply Packet . . . . .	41
4.3 Generating Cyclic Redundancy Check (CRC) Functions with the <code>pycrc</code> Python Script	42
4.4 FaceWorks Verilog Module Declaration ( <code>faw_system.v</code> ) . . . . .	43
4.5 Testbench Header Files . . . . .	43
4.6 Signal Declaration in the SystemC Testbench . . . . .	44
4.7 Interconnecting the Verilated Model and the Testbench Signals . . . . .	45
4.8 Project Folder Hierarchy . . . . .	46
4.9 Verilator Invocation . . . . .	47
4.10 Verilator Directives . . . . .	47
4.11 A Linting Warning Disabled . . . . .	47
4.12 Case Overlap and Case Incomplete Warnings . . . . .	48
4.13 Addition with Narrower RHS Warning . . . . .	48
4.14 Width Mismatch Warning . . . . .	48
4.15 Correction of the Previous Width Mismatch Problem . . . . .	48
4.16 Circular Logic Warning . . . . .	49
4.17 Output from Vcoverage . . . . .	50
4.18 Merging Distinct Coverage Files and Generating Output Statistics . . . . .	50
4.19 Code Coverage Report Excerpt . . . . .	50
5.1 Single-Port Timing Diagram . . . . .	55
5.2 Two-Port Timing Diagram . . . . .	56
6.1 Packet Loss Example Recoveries using NACK's . . . . .	62
6.2 A UDP/CWDP Cycle Accurate Simulator . . . . .	63
A.1 A GTKwave Screenshot Showing FaceWorks Signals . . . . .	69

# List of Tables

1.1	Transistor Density Comparison . . . . .	3
2.1	Media Independent Interface Signals . . . . .	9
3.1	Arbiter FSM States . . . . .	35
3.2	Arbiter Input and Output Signals . . . . .	35
5.1	Implementation Results for the Single-Port FaceWorks . . . . .	52
5.2	Implementation Results for the Two-Port FaceWorks . . . . .	52
5.3	Period Timing Constraints . . . . .	53
5.4	TL-WR841N 10/100 Mbps Maximum Raw Throughput . . . . .	54
5.5	Effective Throughput . . . . .	54
5.6	One Way Delay . . . . .	55
5.7	Throughput Model . . . . .	55
5.8	FaceWorks Throughput Results . . . . .	56
5.9	Measured vs. Modeled Throughput . . . . .	57



# List of Acronyms

<b>ACK</b>	Acknowledge
<b>ARP</b>	Address Resolution Protocol
<b>BRAM</b>	Block Random Access Memory
<b>CRC</b>	Cyclic Redundancy Check
<b>CWDP</b>	Coreworks Datagram Protocol
<b>DCM</b>	Digital Clock Manager
<b>DPLL</b>	Digital Phase-Locked Loop
<b>FIFO</b>	First In First Out
<b>FPGA</b>	Field Programmable Logic Array
<b>FCS</b>	Frame Check Sequence
<b>FSM</b>	Finite State Machine
<b>GPU</b>	Graphics Processing Unit
<b>HDL</b>	Hardware Description Language
<b>IFG</b>	inter-frame gap
<b>ILA</b>	Integrated logic analyzer
<b>IP</b>	Intellectual Property
<b>IP</b>	Internet Protocol
<b>IO</b>	Input-Output
<b>LAN</b>	Local Area Network
<b>LSB</b>	Less Significant Bit
<b>MAC</b>	Media Access Control

## List of Tables

---

**MII** Media Independent Interface

**MSB** Most Significant Bit

**MTU** Maximum Transmission Unit

**NACK** Not acknowledge

**OS** Operating System

**OSI** Open Systems Interconnection

**OWD** One Way Delay

**PC** Personal Computer

**PHY** Ethernet Physical Transceiver

**PLL** Phase-Locked Loop

**RAM** Random Access Memory

**RTT** Round-Trip delay Time

**RHS** Right Hand Side

**SFD** Start Frame Delimiter

**SoC** System-on-Chip

**TCP** Transmission Control Protocol

**UDP** User Datagram Protocol

**VHDL** VHSIC Hardware Description Language

**XST** Xilinx Synthesis Tool



# 1

## Introduction

### Contents

1.1	Scope . . . . .	2
1.2	Problem . . . . .	3
1.3	Solution . . . . .	4
1.4	Objectives . . . . .	4
1.5	Outline . . . . .	4

## 1. Introduction

---

In this chapter the scope of the work is presented, the problem addressed is formulated, the proposed solution is explained, the objectives of the work are stated, and the thesis contents is outlined.

### 1.1 Scope

Nowadays, integrated circuit designers are presented with extremely short design cycles. To deliver on time, companies are required to optimize the design flow. Of the many solutions created, one of the most successful is the reuse of pre-existent and complex blocks, often called Intellectual Property Cores, or simply IP Cores.

A core with well defined and specialized functionality is easier to implement and verify, and can be reused later in other designs. This greatly overcomes some of the complexity of the design and verification process.

Out of this idea a brand new business model has flourished, where IP providers produce verified hardware IP libraries, sometimes associated with software IP libraries, and IP integrators select and integrate multiple IP cores from different vendors on a System-on-Chip (SoC).

Due to the consistent evolution of semiconductor manufacturing processes, integrated circuit manufacturers are raising the gate count per die area year after year, and lowering other metrics such as power consumption and cost per gate [1, p. 24]. Table 1.1 shows the complexity of some large devices from various manufacturers in terms of their transistor area density.

The latest developments in Field Programmable Logic Arrays (FPGAs), placed high-end FPGAs as one of the devices with the highest transistor counts per die area. Launched in September 2011 by Xilinx, the Virtex-7 2000-T presented an astonishing value of 12.85 million transistors per square mm. This is more than the generation of Graphics Processing Units (GPUs) launched later by NVIDIA, in April 2012, the GTX 670 (Kepler), with 12.04 million transistors per square mm. Both devices are fabricated using 28nm technology.

With such high transistor density, several FPGA devices, from mid to high-end, already integrate up to 2 million logic cells and hard blocks for functions like DSP, PCI-E controllers, Ethernet Media Access Controls (MACs), memory controllers, DCM/PLL, BRAMs, etc. These highly heterogeneous devices allow engineers not only to use FPGAs as a prototyping platform but also as the target platform.

As FPGAs present themselves as flexible and cost effective platforms to test, verify and produce IP cores, challenges arise on how to properly communicate independently with each IP core in the FPGA. IP cores need to be observed, configured, tested and debugged. Complex hardware systems demand complicated and concurrent testing and debug methods, often requiring expensive equipment to be used.

Table 1.1: Transistor Density Comparison

Device	Millions of Transistors	Die Area ( $mm^2$ )	Density
GPU			
Nvidia GTX 670 (2012 - 28 nm) [2]	3540	294	12.04
AMD Tahiti RV1070 (2011 - 28nm) [3]	4312	365	11.81
Nvidia GF100 Fermi(2010 - 40 nm) [3]	3200	526	6.08
AMD Caymn RV870 (2010 - 40 nm) [3]	2154	334	6.44
AMD RV790XT (2008 - 55nm) [3]	959	282	3.40
NVIDIA GT200 Tesla (2008 - 55nm) [3]	1400	576	2.43
CPU			
Intel Ivy Bridge (22 nm -2012) [4]	1400	160	8.75
Amd Trinity (32nm - 2012) [5]	1178	228	5.17
Intel Xeon 2690EP (32 nm -2011) [6]	2260	416	5.43
Amd Bulldozer 8 Cores (32 nm - 2011) [7]	1200	315	3.81
FPGA			
Xilinx Virtex-7 2000 T (28nm - 2011) [8]	6800	529	12.85

To solve this problem, the company Coreworks SA created and patented a technology [9, US 2008/0288652], [10, EP 2003571/A2] that uses a simple PC and the office network to exercise multiple cores in the FPGA. The technology has been called Core Access Networks® and comprises the FaceWorks IP core, which is the gateway between the IP Cores under test and a software program (test driver) running on the PC.

Prior devices for observation, configuration, testing and debugging on-chip cores resorted to already existing communication protocols. Most of them used low-speed serial protocols with low pin counts, such as I2C, SPI or JTAG, which was initially created to test printed circuit boards and later extended to chips.

FaceWorks implements the User Datagram Protocol (UDP)/Internet Protocol (IP)/Ethernet protocol stack in hardware, dispensing with an embedded processor and a software protocol stack. In addition, the Coreworks Datagram Protocol (CWDP) was created to implement test and debug functions on top of the UDP/IP stack. Other functions such as data streaming can also be performed with this technology. This solution represents an almost costless, high-speed and versatile test and debug mechanism.

## 1.2 Problem

The implementation of FaceWorks that existed when this work started used only a single UDP port to communicate with the FPGA board. The main problem of having a single port is that, to exercise IP modules concurrently, the test driver application becomes very difficult to write. This is the problem that this work solves.

### 1.3 Solution

To solve the problem formulated in the previous section, it has been decided to increase the number of UDP ports supported by FaceWorks, in order to allow multiple software processes or threads to independently and concurrently stimulate and observe multiple Intellectual Property (IP) cores in the system.

### 1.4 Objectives

The main objectives of this thesis are the following:

1. Upgrade the FaceWorks hardware so that it can support multiple and concurrent channels by means of multiple UDP ports. The upgrade is done using the Verilog language, which implies converting the original code from VHSIC Hardware Description Language (VHDL) to Verilog.
2. Write a SystemC verification environment (testbench) that uses the FaceWorks SystemC model generated by the Verilator tool.
3. Write an application that generates and sends test patterns for the multiple UDP ports, receives the responses and checks the results. This application can communicate indistinguishably with either the FPGA board or the SystemC testbench.

### 1.5 Outline

This thesis is structured in the given form:

#### Chapter 2

The main concepts of the protocols and interfaces used by FaceWorks are presented. The existing FaceWorks architecture is also presented.

#### Chapter 3

The development methodology for both the software and hardware components of this work is presented.

#### Chapter 4

The architecture of the application that uses Verilator to create a FaceWorks simulator is presented.

#### Chapter 5

Implementations results are presented and compared with the previous implementation. A characterization of the network is performed and an analysis of the network performance of FaceWorks is presented.

## Chapter 6

Conclusions are drawn. Future modifications and alternative implementations are discussed.



# 2

## Background

### Contents

---

2.1	Media Independent Interface . . . . .	8
2.2	Media Access Control . . . . .	10
2.3	Address Resolution Protocol . . . . .	12
2.4	Internet Protocol . . . . .	14
2.5	User Datagram Protocol . . . . .	16
2.6	Coreworks Datagram Protocol . . . . .	18

---

## 2. Background

In this chapter the FaceWorks IP core is introduced, and its interfaces and protocols are explained. The existing FaceWorks architecture is shown in Figure 2.1. In the following sections, the implementations of the Media Independent Interface (MII) interface, data link, network and transport layers are presented.

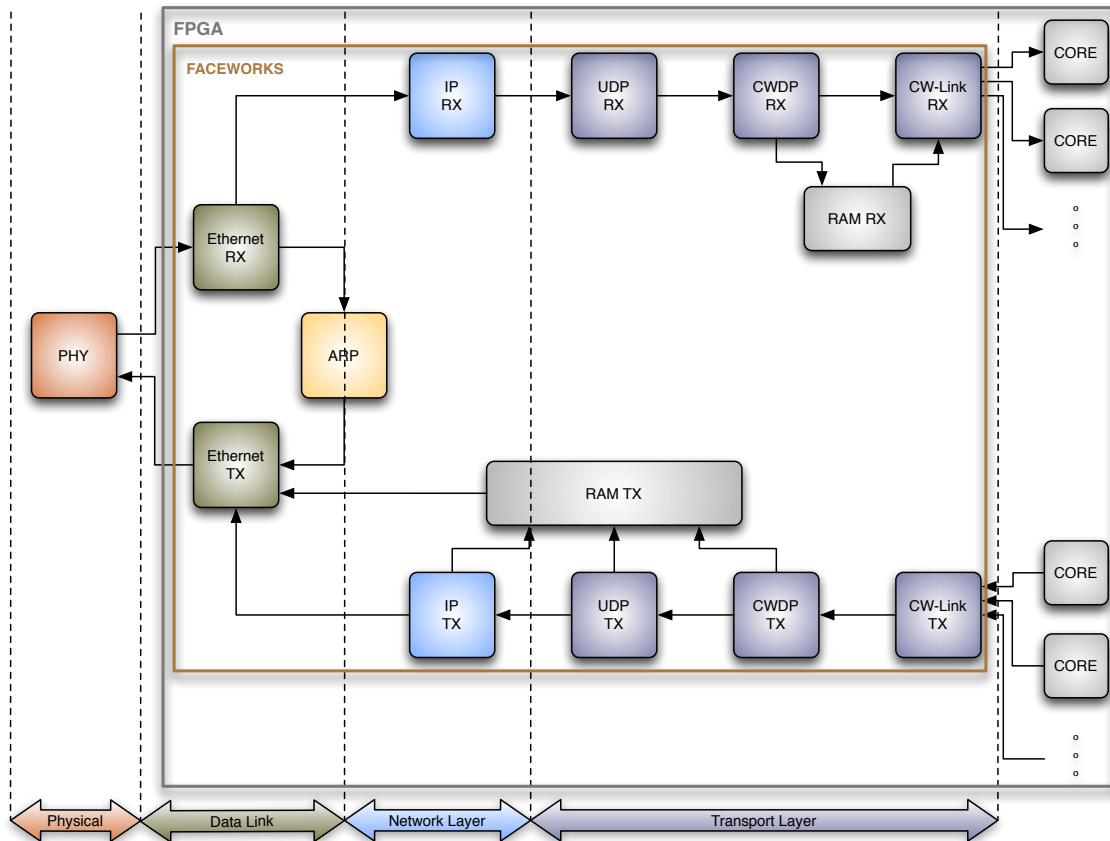


Figure 2.1: FaceWorks Block Diagram

### 2.1 Media Independent Interface

The MII[11] is a well known industry standard for the interface between the MAC and the Ethernet Physical Transceiver (PHY). In terms of the OSI layer model, it implements the interface between the physical layer and the data link layer. MII is used to interconnect the FaceWorks core in the FPGA and the PHY chip on the same board, as depicted in Figure 2.2. The MII signals of the FaceWorks core are presented in Table 2.1.

Both TX\_CLK and RX\_CLK are 25 MHZ clock signals, required for a PHY operating at 100 Mbps. The signal RX\_ER is not used. Instead the Cyclic Redundancy Check (CRC) field provided by the MAC layer is used to check the frame content. The signal `mdio` is in high-impedance as both the managing signals are not used.



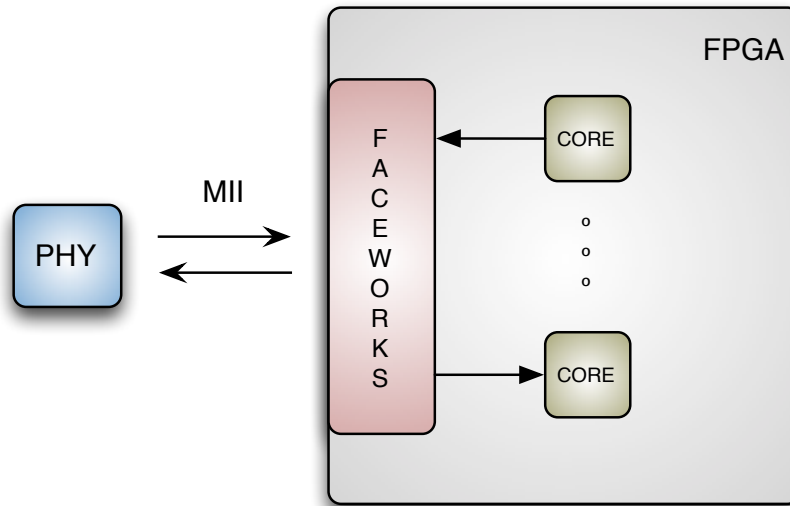


Figure 2.2: Interconnection of PHY and FaceWorks using MII

Table 2.1: Media Independent Interface Signals

Signals	Direction	Description
Transmit		
TX_CLK	input	Clock signal reference for the transmit signals
TX_EN	output	MAC is presenting data on the MII for transmission
TX_D[3:0]	output	Transmit Data
TX_ER	output	MAC Signals the transmission of a coding error on the frame
Receive		
RX_CLK	input	Clock signal reference for the receive signals
RX_DV	input	MII is presenting data to the MAC
RX_D[3:0]	input	Receive Data
RX_ER	input	PHY signals the transmission of error on the frame
Management		
MDC	input	Management data clock
MDIO	output	Management data input/output

## 2. Background

---

### 2.1.1 Data Stream

Packets transmitted through the MII, are framed as shown in Figure 2.3. Below follows an explanation of each field.

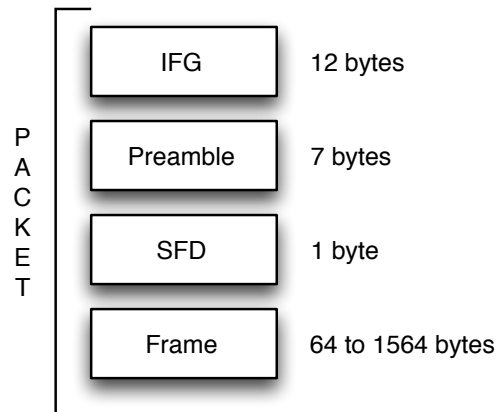


Figure 2.3: MII Packet

**IFG** The inter-frame gap (IFG) is a period of 12 bytes in which the `TX_EN` or the `RX_EN` signals are asserted low to produce an idle time between the previous packet and the current packet. This allows both the PHY and the MAC to process the previous packet, and prepare for the reception of the new packet.

**Preamble** The preamble consists of a sequence of 56 bits of alternating 1's and 0's <sup>1</sup>, in order to allow the receiver PHY Digital Phase-Locked Loop (DPLL) to lock.

**SFD** The Start Frame Delimiter (SFD) consists in the bit sequence 10101011 <sup>1</sup>, marks the start of the packet and the end of the preamble.

**Frame** Contains the actual payload that is transmitted through the MII.

## 2.2 Media Access Control

The Media Access Control (MAC)[12] protocol together with the MII interface implement the Ethernet data link layer and the interface to the physical layer, respectively.

From the packet format defined previously for the MII interface, the MAC encapsulates the data payload with a 14-byte header and a 4-byte CRC trailer, as shown in Figure 2.4. Below follows an explanation of each field.

---

<sup>1</sup>The stream is transmitted from left to right.

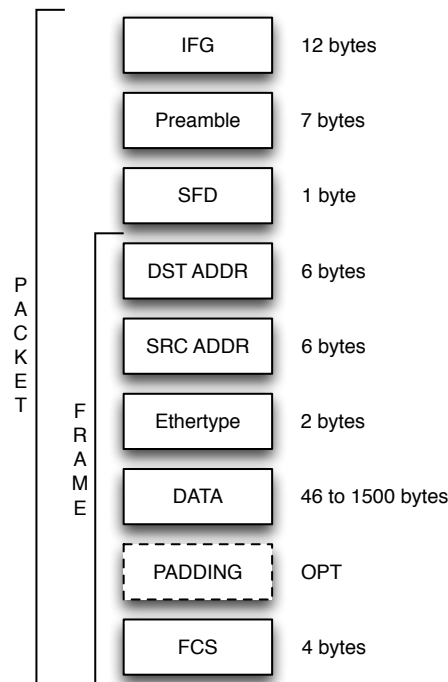


Figure 2.4: Ethernet Packet Structure with the MAC Protocol

**DST ADDR** The destination address is a 6-byte field that specifies the MAC adapter(s) for which the packet shall be delivered, through the use of an unicast or multicast address.

**SRC ADDR** The source address is a 6-byte field that specifies the unique MAC adapter from which the packet is sent.

**Ethertype** This is a 2-byte field<sup>2</sup> field that identifies the type of protocol being carried on the payload.

**Data** This is the field that carries the inner protocol payload.

**Padding** This is an optional field necessary when the Data field is smaller than the specified size of 46 bytes.

**FCS** This is a 4-byte field that contains the Cyclic Redundancy Check (CRC) of the MAC frame. The CRC is calculated over all fields of the MAC frame, except the Frame Check Sequence (FCS) field. The CRC uses the 0x04c11db7 polynomial.

The FaceWorks implementation of the MII interface and MAC layer consists of two logic blocks, an Ethernet receiver and an Ethernet transmitter, as shown in Figure 2.1.

<sup>2</sup>Type for ARP is 0x0806, for IP is 0x0800

## 2. Background

---

The receiver block is responsible for removing the preamble, detecting the SFD, performing packet filtering based on the destination MAC and Ethertype, and checking the CRC to validate the data.

The inverse operation is performed by the transmitter block: it adds the preamble to the payload, inserts the SFD, the Ethernet header, and calculates the trailing CRC.

As imposed by the specification, the MII interface clock has a period of 40 ns. However, the internal FaceWorks clock period is often faster. Since two clock domains exist, two asynchronous First In First Outs (FIFOs) are necessary to buffer the data in each direction.

### 2.3 Address Resolution Protocol

The Address Resolution Protocol (ARP)[14] is used for resolving network layer addresses into data link layer addresses. Since IPv4 over Ethernet is being used, the correspondence between an IP address and a MAC address is established. It can also be used for different types of network layer and data link layer protocols.

When a computer A wants to send a packet to a computer B in the Local Area Network (LAN), it needs to identify it with the IP address and the MAC address of computer B, to unequivocally tell the computer that is supposed to get this packet.

Assuming that computer A knows the IP address of computer B, it sends an ARP Request to all computers in the LAN, to inquire which computer possesses that IP address. Upon receiving the ARP Request, computer B replies with an ARP Reply packet, which causes computer A to learn the MAC address of computer B, and finally send the packet to computer B. To avoid sending an ARP Request every time a packet needs to be sent, computer A holds a simple ARP cache, where it stores IP/MAC address pairs. Figure 2.5 shows how an ARP packet is structured. This packet format is used for both the ARP Request and for the ARP Reply. Each field of an ARP packet is explained next.

0	7	8	15	16	31
Hardware Type (HTYPE)			Protocol Type (PTYPE)		
HLEN		PLEN		Operation (OPER)	
Sender hardware address (SHA) [bytes 0-3]					
Sender hardware address (SHA) [bytes 4-5]			Sender protocol address (SPA) [bytes 0-1]		
Sender protocol address (SPA) [bytes 2-3]			Target hardware address (THA) (bytes 0-1)		
Target hardware address (THA) [bytes 2-5]					
Target protocol address (TPA) [bytes 0-3]					

Figure 2.5: ARP Packet

**HTYPE** The Hardware Type specifies the data link layer protocol type; for Ethernet this value is 0x0001.

**PTYPE** The Protocol Type specifies the network layer protocol; for IPv4 it has the value 0x0800.

**HLEN** The Hardware Length specifies the data link layer address size; for Ethernet this value is 0x06 (length of the MAC address).

**PLEN** The Protocol Length specifies the network layer address size; for Ethernet this value is 0x04.

**OPER** The Operation field specifies the type of the ARP packet; the value is 1 for an ARP Request, 2 for an ARP Reply.

**SHA** The Sender Hardware Address contains the sender MAC address.

**SPA** The Sender Protocol Address contains the sender IP address.

**THA** The Target Hardware Address contains the target MAC address; this field is ignored in ARP Requests.

**TPA** The Target Protocol Address contains the target IP address; this field is ignored in ARP Requests.

The FaceWorks ARP implementation consists in a single-entry ARP table, to create the correspondence between the IP address and the MAC address of the host that drives FaceWorks. The ARP table is connected to both the Ethernet RX and Ethernet TX blocks, as shown in Figure 2.1.

Upon reception of an ARP Request, the Ethernet RX block forwards the packet to the ARP block which generates an ARP Reply packet and requests the Ethernet TX block to transmit it to the network.

When the Ethernet TX block is constructing an Ethernet packet to be sent, it provides the destination IP of the host to the ARP logic. If the IP is registered in the table, the ARP block outputs the host MAC address; otherwise the ARP block generates an ARP Request, which is sent by the Ethernet TX block. Later, the respective ARP Reply will be processed by the Ethernet RX block, which forwards the IP and MAC address to the ARP block. After this the ARP table holds the necessary information to process the current and future Ethernet TX requests to the same host.

## 2. Background

Figure 2.6 exemplifies how values are assigned to the ARP packet fields: computer A with IP address 192.168.0.22 and MAC address 00:26:9e:e2:e3:14 broadcasts an ARP Request to find IP address 192.168.0.133, and receives a reply from FaceWorks containing its MAC address.

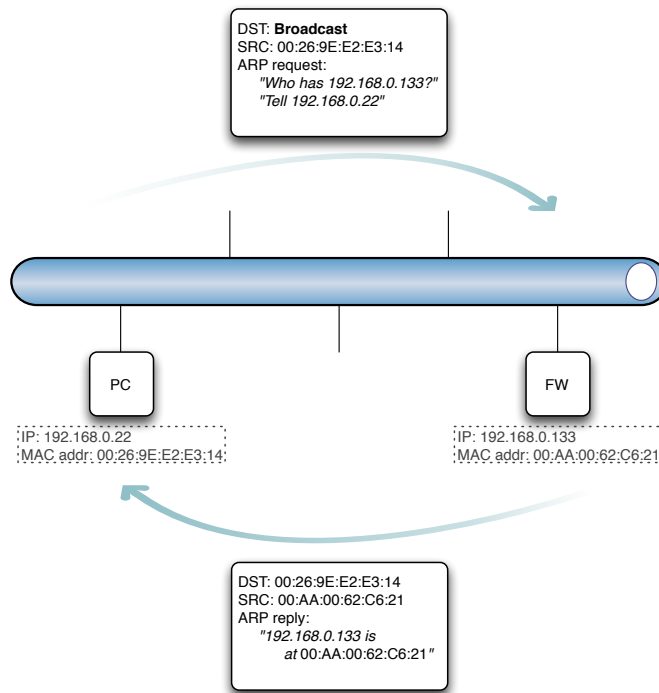


Figure 2.6: Address Resolution using ARP

## 2.4 Internet Protocol

The Internet Protocol (IP)[15] main purpose is to provide an abstraction layer that hides differences on the data link layer implementation. It offers a uniform addressing and routing scheme, and a fragmenting mechanism so that different Maximum Transmission Unit (MTU) values can be supported across different data link layer technologies. FaceWorks only supports IPv4 packets; therefore, IPv6 packets are not discussed here. An IP packet consists of header and payload. Figure 2.7 shows the format of an IPv4 packet header. The meaning of each field is explained below.

0	3	4	7	8	13	14	15	16	18	19	23	24	31
Version		IHL		DSC		ECN		Total Length					
Identification						Flags		Fragment Offset					
Time to Live			Protocol					Header Checksum					
Source IP Address													
Destination IP Address													
Options (Optional)												Padding	

Figure 2.7: IPv4 Packet Header

**Version** This 4-bit field identifies the version number of the Internet Protocol, only versions v4 and v6 are defined. For IPv4 this value is 4.

**IHL** The Internet Header Length is also a 4-bit field that contains the length of the packet header in 32-bit words, and points to the beginning of the data. The smallest valid value is 5 and the highest value is 15 (corresponds to a header length of 60 bytes).

**DSC** The Differentiated Services Codepoint is a 6-bit field used for describing the intended forwarding behavior. This is mainly used for real-time applications, and is not used in FaceWorks.

**ECN** The Explicit Congestion Notification is a 2-bit optional field, which is an extension to TCP/IP used for end-to-end congestion notification. This field is not supported by FaceWorks.

**Total Length** This is the total length of the IP packet. It is a 16-bit field, so the maximum size of an IP datagram is 65535 bytes. The minimum-length of a packet is 20 bytes (size of header without data). The largest datagram size that a IP-enabled computer is required to reassemble is 576 bytes.

**Fragment ID** The destination computer uses this identifier and the sender's address to identify fragments of IP datagrams, so that the original datagrams are reconstructed at the destination computer. FaceWorks does not support IP fragmentation.

**Flags** The flag field is used for fragmented IP packets. It contains two flags: Don't Fragment (DF) and More Fragments (MF). The DF bit shows that the datagram must not be fragmented, even if it cannot be forwarded further. The MF bit shows more fragments exist in this IP packet. Thus, the last packet of a fragmented IP datagram has MF set to 0. Both these flags are not supported in the FaceWorks IP implementation.

**Fragment Offset** Specifies where in reference to the beginning of the entire datagram the present fragment has to be ordered. This information is essential to reassemble the original packet from the individual fragments in the case fragments arrive out-of-order. This is also not supported by the FaceWorks IP implementation.

**TTL** The Time To Live is an 8-bit field that is used to prevent datagrams from going in circles in the internet. It implements a counter that for each router on the path is decremented by at least one. If the field reaches value 0, then the packet is discarded.

## 2. Background

---

**Protocol** An 8-bit field that describes the protocol used in the payload of the IP datagram.<sup>3</sup>

**Header Checksum** This field contains a 16-bit checksum of the IP packet header. It is used for error-checking the IP header. When a router receives an IP packet it calculates the checksum and compares it with the value in this field. If the values match the TTL is decremented by 1 and the checksum has to be updated since the header content has been updated. Since the payload data is not verified by this checksum, the payload protocols have their own checksum.

**Sender Address** This field contains the 32-bit IP address of the sender computer.

**Destination Address** This field contains the 32-bit IP address of the destination computer.

**Options** This field is used for user data, and must comply with a standard format not discussed here. Since the header length must be a 32-multiple, the user is required to insert padding bits. This field not used or supported by FaceWorks.

The implementation of the IP protocol in FaceWorks uses two blocks, the IP RX block and the IP TX block, as depicted in Figure 2.1.

The IP RX block implements the IP protocol for the receive packets. It processes the IP packets that are received from the MAC layer, according to the values of their IP header version, destination IP address and protocol type. It calculates the checksum to verify the header and forwards the payload data to the next block, the UDP RX block.

For transmit packets the IP TX block calculates the header checksum and writes the IP header and checksum in the RAM TX block, which is where the packet is being formed. Finally, it requests the Ethernet TX module to send the packet.

## 2.5 User Datagram Protocol

The User Datagram Protocol (UDP)[16] provides an unreliable, connectionless transport datagram service. UDP uses the Internet Protocol as the underlying protocol.

UDP is suitable for this work because error checking is not necessary, and retransmission of lost packets is implemented by a higher application protocol. This way the overhead of a TCP session is avoided, and a much simpler implementation on the hardware is obtained.

### 2.5.1 UDP Packet Header

A UDP packet consists of header and payload. Figure 2.8 shows the format of a UDP packet header. The header fields are briefly described below.

---

<sup>3</sup>UDP is 17



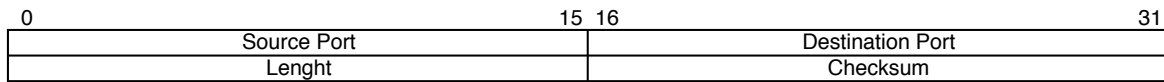


Figure 2.8: UDP Packet Header

**Source Port** This is a 16-bit field which identifies the port used by the sender process.

**Destination Port** This is a 16-bit field that identifies the destination port used by the receiver process.

**Length** This is a 16-bit field, which contains the size in bytes of the packet header and payload combined.

**Checksum** This a field used for verifying the contents of the header and data. The checksum is calculated over a pseudo-header, the UDP header and the data. Padding may be necessary in order to have multiples of two bytes.

### 2.5.2 Implementation

The FaceWorks implementation for the transport layer is also composed of two blocks, the UDP RX and the UDP TX blocks, as shown in Figure 2.1.

The UDP RX block filters out UDP messages that are not being sent to the UDP port being used for CWDP communication. For messages addressed to the UDP port being used for CWDP communication, the UDP RX block removes the UDP header and forwards the UDP payload (CWDP packet) to the CWDP RX module.

The UDP TX block inserts the UDP header in the RAM TX buffer, calculates the checksum assuming the pseudo-header is attached upfront, and forwards the transmission of the UDP packet to the IP TX module.

## 2.6 Coreworks Datagram Protocol

The Coreworks Datagram Protocol (CWDP)[17] is a proprietary protocol, which runs over UDP, and was created to communicate with the on-chip cores using the FaceWorks core. Each CWDP packet is sent as a UDP payload, as shown in Figure 2.9.

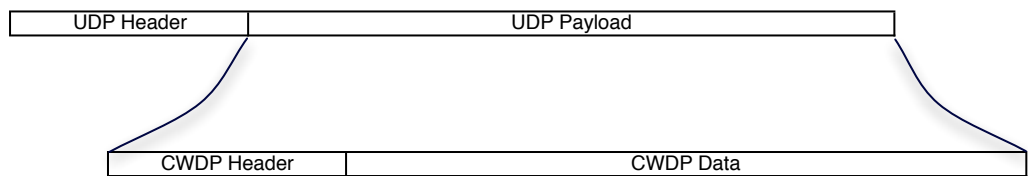


Figure 2.9: CWDP Encapsulation in the UDP Payload

For this project only a subset of the FaceWorks commands is presented, since there are several features that have not been used in this work.

### 2.6.1 CWDP Packet

Figure 2.10 shows how a CWDP packet is structured. The fields are briefly described below

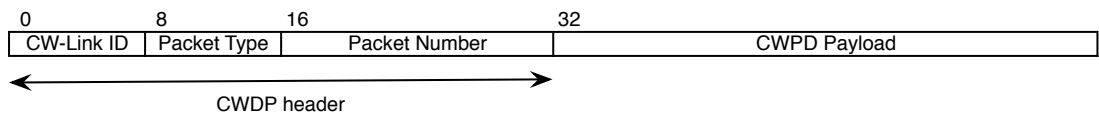


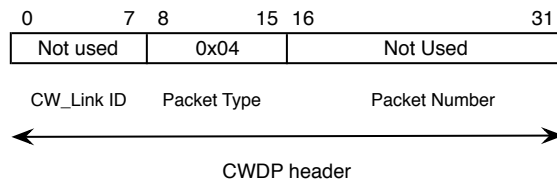
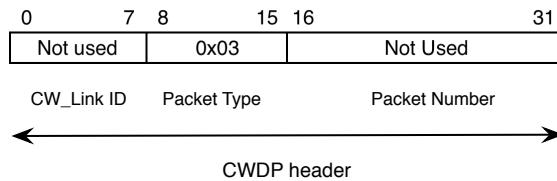
Figure 2.10: CWDP Packet

**CW-Link ID** This field identifies the destination core interface for which the packet data is intended.

**Packet Type** This field identifies the CWDP packet type; not all available packet types are used in this work.

**Packet Number** This field is a sequence number for data packets. It is used to implement reliable transmission. The Packet Number is incremented for every packet acknowledged by the receiver. Only packets with the expected Packet Number are acknowledged by the receiver.

**Payload** Contains the data to be delivered to the cores, or parameters for FaceWorks.



## 2. Background

---

**CORE\_DATA** (packet type 0x01) This packet type is used to send data between systems and cores. In Figure 2.13 a CWDP CORE\_DATA packet is shown. The packet payload is composed of several CW-Link words, each word is 32-bit long and each CORE\_DATA packet can contain up to 360 CW-Link words.

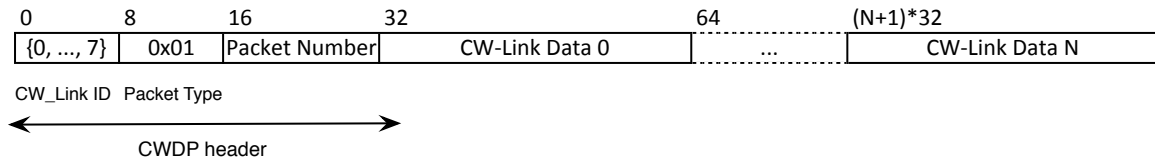


Figure 2.13: CWDP CORE DATA Packet

**ACK** (packet type 0x02) This packet type is used to acknowledge received packets. In Figure 2.14, the structure of a CWDP ACK packet is shown. Received CWDP packets, except for the Acknowledge (ACK) packet itself, trigger a reply with an ACK packet. The packet number of the transmitted ACK matches the packet number of the received packet when a CORE DATA packet is received. If the received packet is a SET\_CORE\_ACCESS packet then the packet number of the ACK packet is set to zero.

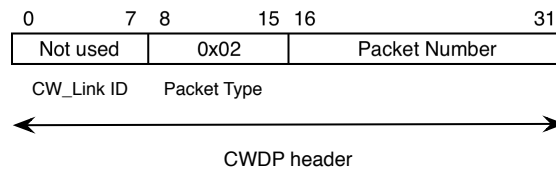


Figure 2.14: CWDP ACK Packet

### 2.6.3 Implementation

The CWDP application layer is implemented using 4 logic blocks (CWDP RX, CWDP TX, CW-Link RX and CW-Link TX), as shown in Figure 2.1.

The CWDP RX block implements the CWDP protocol for the received packets. It decodes the type of the received packet and performs the instructed action. The payload of CORE\_DATA packets is placed in the RAM RX buffer. After the reception of a CWDP packet it requests the CWDP TX to send the acknowledge packet. If the CWDP packet is a CORE DATA packet it forwards the data to CW-Link RX block in order to be delivered to the addressed IP Core. An interaction diagram between the CWDP receive and transmit sides is shown in Figure 2.15.

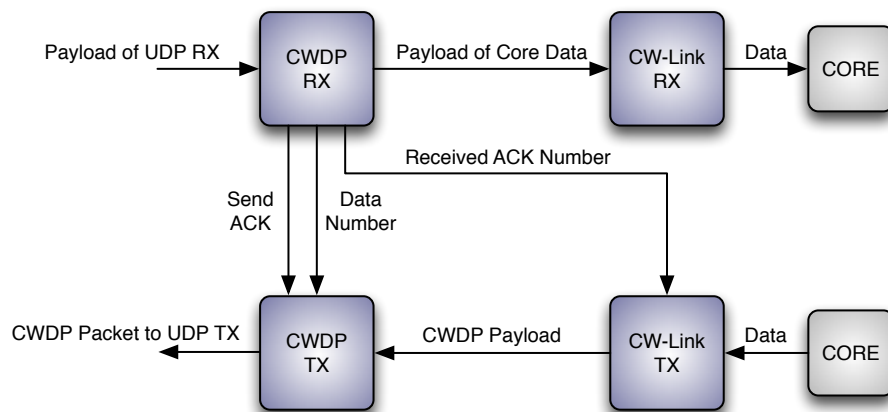


Figure 2.15: CWDP Layer

The CWDP TX block receives data from the CW-Link TX block and sends the data to the UDP TX module by means of the RAM TX buffer. Then it waits for the ACK packet of the sent packet, which will arrive in the CWDP RX block. When the ACK packet arrives, the CWDP RX block presents the sequence number to the CWDP TX block, so it can verify that it matches the sequence number of the packet previously sent.

### 2.6.4 CW-Link Interface

FaceWorks uses CW-Link interfaces[17] to send data or commands to and from the cores inside the chip. In Figure 2.16 the CW-Link interfaces between FaceWorks and a few cores are shown.

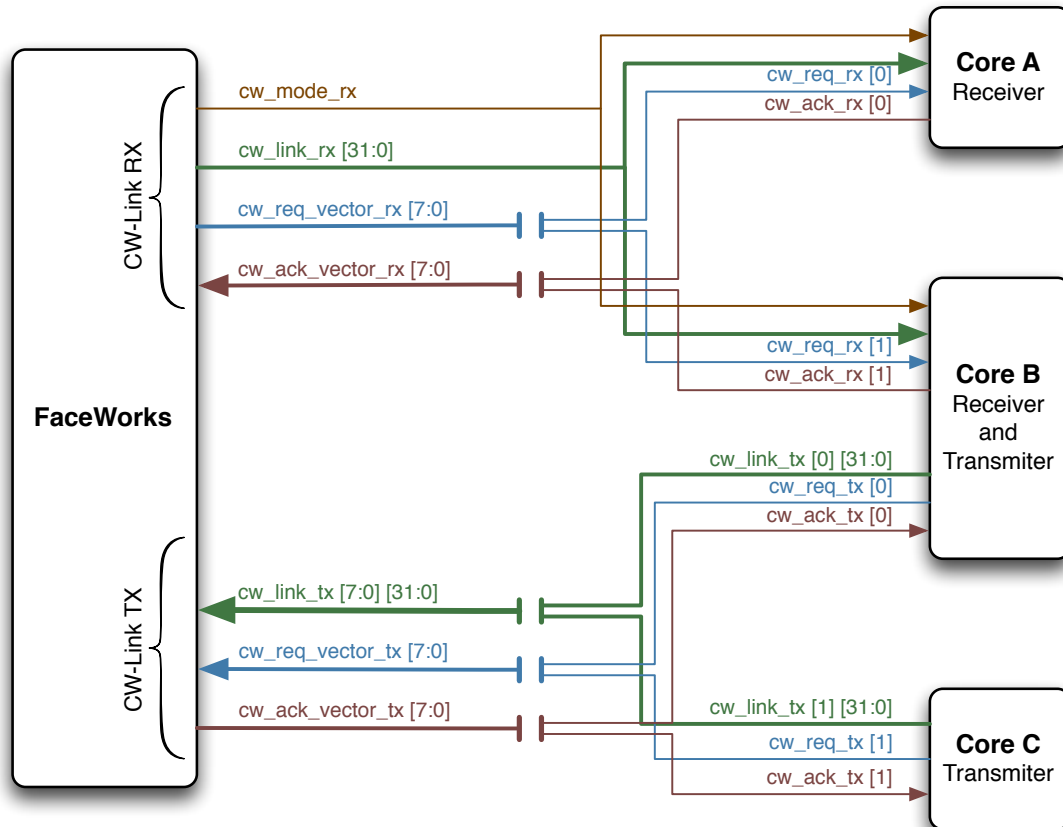


Figure 2.16: CW-Link Interface

The CW-Link interfaces specifies four logic signals: request (`req`), acknowledge (`ack`), data (`link`) and mode (`mode`). FaceWorks is the master of the CW-Link RX interface and is a slave of the CW-Link TX interface. The mode signal is only used in the CW-Link RX interface to distinguish between data and commands for the IP cores. The CW-Link RX may be interconnected to eight different IP cores.

As depicted in Figure 2.16 both the receive data bus and mode signal are shared between the receiving cores, and there is a `req/ack` pair for each of the 8 cores; only one of the `req` signals may be active at any instant. There is one transmit data bus and one `req/ack` pair for each core.

A timing diagram of CW-Link RX transactions is shown in Figure 2.17. The communication uses simple handshaking: a single bit from the `req` signal vector goes high to identify the destination IP core while the data bus holds a valid value until the respective `ack` signal goes high.

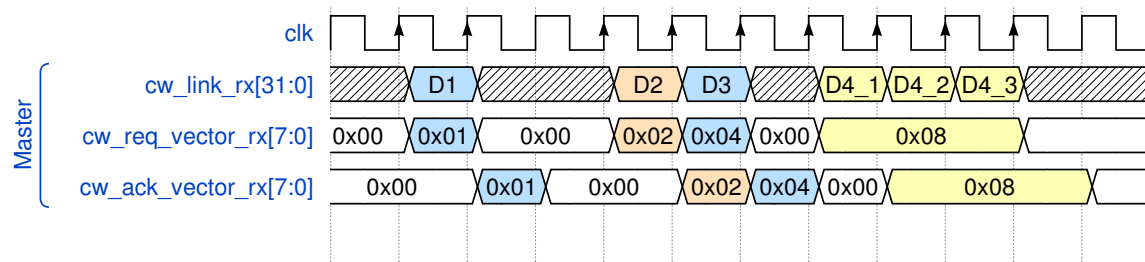


Figure 2.17: CW-Link RX Transactions

CW-Link also supports burst transfers by keeping the request signal high and sending data words consecutively as long as the `ack` signal is kept high. An example burst transfer is depicted in yellow in Figure 2.17<sup>4</sup> and consists of 3 data words for core 4. The same rationale is used for the CW-Link TX interface.

<sup>4</sup>This timing diagram has been constructed using the javascript web application Wavedrom available at: <http://wavedrom.googlecode.com>





# 3

## Design

### Contents

---

3.1 Face-Test Application . . . . .	26
3.2 Hardware Implementation . . . . .	31

---

### 3. Design

---

In this chapter the implementation of multiple UDP ports for FaceWorks is presented. The software components and the hardware modifications effected to support two distinct ports are explained. The same method can be used for more than two ports.

During the development loopback wires have been used, to directly connect the CW-Link RX block to the CW-Link TX block, as shown in Figure 3.1. With the CW-Link loopback, there is no need for a stimulus generator for the CW-Link TX interface. CORE\_DATA packets are sent from a host computer and are delivered back to the host. This way packets that are sent and received back can be compared to detect possible differences.

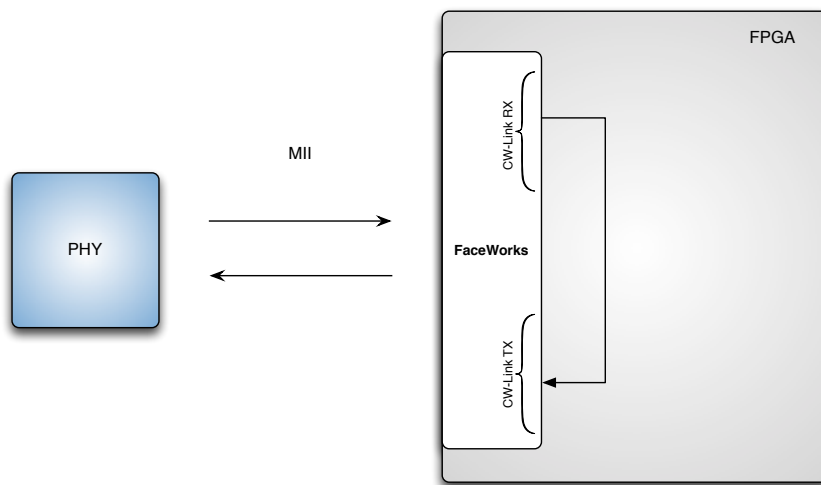


Figure 3.1: CW-Link Loopback

## 3.1 Face-Test Application

The FaceWorks test application (Face-Test) is a C language application that implements the CWDP protocol. The application has been tested on the pre-existing FaceWorks architecture and on the new FaceWorks architecture.

### 3.1.1 Basic Functions

The two basic functions used in the Face-Test application are the `CWDP_receive_packet()` and the `CWDP_send_packet()` functions. These functions create an abstraction layer which hides the CWDP details. They are used for sending / receiving packets to / from FaceWorks, and can be called for different sessions/connections inside a user process.

The `CWDP_receive_packet` function (Figure 3.2) reads UDP packets from a socket descriptor `fd` and places the content in buffer `buf_in`. If the received packet is an ACK packet the value of `core_out` (sequence number of the last packet sent) is compared with the sequence number of the received ACK packet. If they do not match the `CWDP_receive_packet()` function returns the error

code `P_ERROR`; otherwise the function returns the type of packet received. If a `CORE_DATA` packet is received the `ack_out` variable is updated with the sequence number of the received `CORE_DATA` packet so a `ACK` packet can be sent with the correct value.

```
int CWDP_receive_packet (int fd, struct sockaddr_in server_addr, byte *buf_in,
u_int16_t *core_out, u_int16_t *ack_out);
int CWDP_send_packet (int fd, struct sockaddr_in addr, struct sockaddr_in server_addr, byte *buf_out,
byte * buf_in, int buf_size, e_cwdp cwdp_type, u_int16_t *core_out, u_int16_t *ack_out);
```

Figure 3.2: CWDP Send/Receive Functions

The `CWDP_send_packet` function (3.2) sends a UDP packet to a socket descriptor `fd`, using as destination the FaceWorks IP address `addr`.

If the type of packet sent requires an acknowledge from FaceWorks, the `CWDP_send_packet()` function internally calls to the `CWDP_receive_packet()` function to get the packet, and then verifies if the sequence number is the expected one.

#### 3.1.2 Test Application

The Face-Test application is a simple C application that consists in sending `CORE_DATA` packets with random data from a PC to a network connected FPGA with a FaceWorks core.

After defining the socket settings the application starts by sending a `SET_CORE_ACCESS` packet as depicted in Figure 3.3 to establish the PC controlling the FaceWorks core.

```
/*Send Core Access and receive Ack*/
buf_size=0;
CWDP_send_packet (fd,addr,server_addr,buf_out,buf_in,buf_size,P_SET_CORE_ACCESS,&core_out,&ack_out);
```

Figure 3.3: Send Core Access and Wait for Acknowledge

As Ethernet is a best-effort service, a packet sent by the PC can fail to be delivered to FaceWorks, and the acknowledge sent by FaceWorks can fail to be delivered to the PC.

To resolve both these situations it is needed that the `CWDP_receive_packet()` function have a timeout when reading data from the socket. When the `CORE_ACCESS` packet sent from the PC is lost in the network, FaceWorks will not reply with the `ACK` packet.

### 3. Design

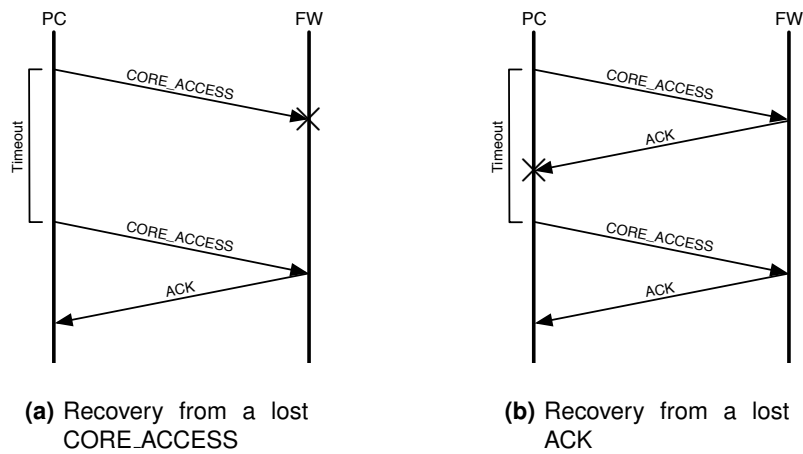


Figure 3.4: Recovering from Lost Packets

When the PC is waiting to receive an ACK packet from FaceWorks, if a timeout did not exist, the PC would block. Therefore, after a timeout, the `CWDP_receive_packet()` function returns `P_ERROR` and the `CORE_ACCESS` packet is retransmitted. This situation is depicted in Figure 3.4(a).

The same procedure applies for the loss of the ACK packet, as depicted in Figure 3.4(b). Assuming the ACK packet transmitted by FaceWorks is lost in the network, the PC waits for it until the `CWDP_receive_packet()` function times out and returns `P_ERROR`. After that, the `CORE_ACCESS` packet is retransmitted.

After the PC sends the `SET_CORE_ACCESS` packet to FaceWorks successfully, it can start to send `CORE_DATA` packets. To accomplish this the `CWDP_send_packet()` function is called as shown in Figure 3.5.

```
/*Send Core Data and receive Ack*/  
CWDP_send_packet(fd,addr,server_addr,buf_out,buf_in,buf_size,P_CORE_DATA,&core_out,&ack_out);
```

Figure 3.5: Send Core Data and Wait for Acknowledge

Now the PC expects to receive the same CORE\_data packet it sent to FaceWorks. To accomplish this the `CWDP_receive_packet()` function is called inside a `do while` loop until a `CORE_DATA` packet type is received (Figure 3.6).

```
/*Read Core Data and send Ack*/
do{
    do{
        memset((void*)&buf_in,(unsigned char)0,sizeof(buf_in));
    }while(CWDP_receive_packet(fd,server_addr,buf_in,&core_out,&ack_out)!=P_CORE_DATA);

    CWDP_send_packet(fd,addr,server_addr,buf_out,buf_in,buf_size,P_ACK,&core_out,&ack_out);

    /*Compare sent packet with received packet*/
    if(memcmp(&buf_out[4],&buf_in[4],1444)==0)
    {
#ifdef PREF_TEST
        printf("Data Match!!\n");
#endif
        (u_int16_t)(core_out)++;
    }
    else
    {
        printf("Invalid Data!!\n");
    }
#ifdef PREF_TEST
    printf("Core seq number: %d Ack seq number: %d\n",core_out,ack_out);
#endif
}while(memcmp(&buf_out[4],&buf_in[4],1444)!=0);
```

Figure 3.6: Read Core Access and Send Acknowledge

### 3. Design

---

When the CORE\_DATA packet is received from FaceWorks, the PC replies with an ACK packet. If this ACK packet is not delivered, the PC will not know of this and will transmit the next CORE\_DATA packet.

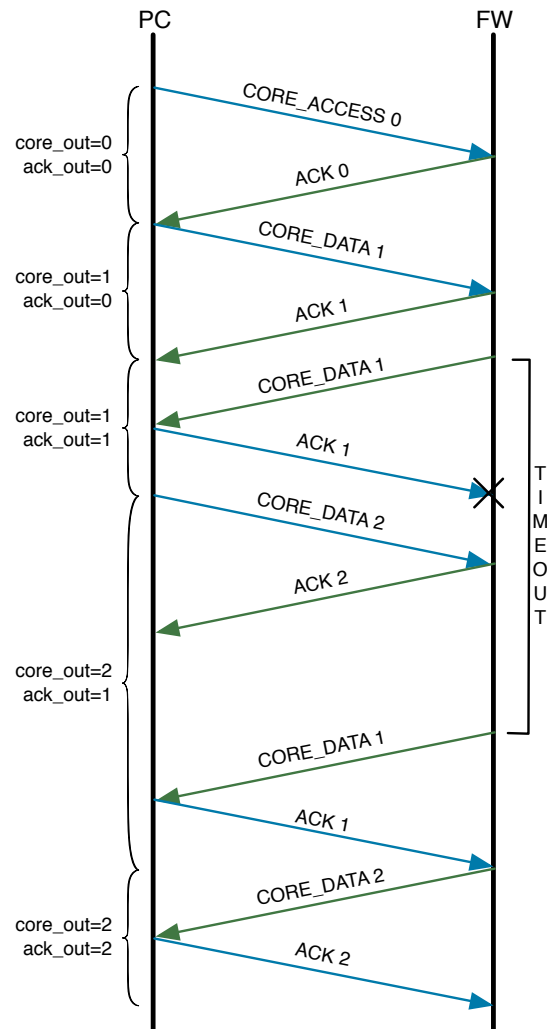


Figure 3.7: Facework Recovery from a Lost Packet

FaceWorks assumes the previous CORE\_DATA packet sent was not received by the host, so after a timeout period it resends the same CORE\_DATA packet. A recovery from such failure is depicted in the network diagram in Figure 3.7.

## 3.2 Hardware Implementation

The new architecture is presented in Figure 3.8. Compared to the original architecture, the CWDP\_Link modules are duplicated on both the reception and transmission sides. The RAM\_RX block is also duplicated in order to receive and store two consecutive packets with different destination ports. An arbiter is added to decide which of the CW-Link TX is granted access to the medium.

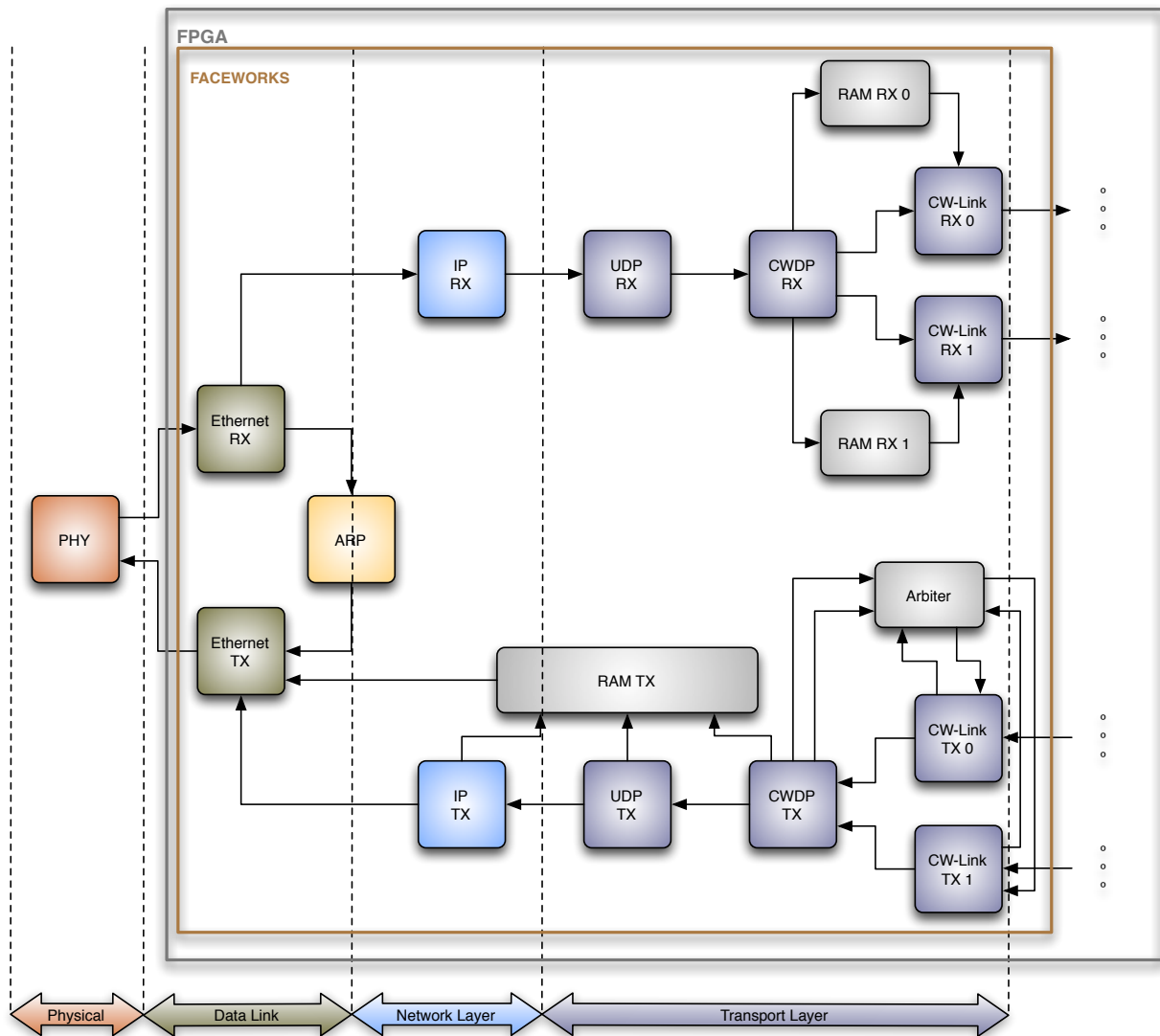


Figure 3.8: Two-Port FaceWorks Block Diagram

#### 3.2.1 Modified Blocks

In this implementation FaceWorks supports a single sender IP address and listens to two hardwired UDP ports, whose numbers differ by 1. For example, ports 1234 and 1235. The following paragraphs describe the modifications effected on the original hardware blocks.

**UDP RX** The UDP RX block original implementation filtered out the UDP packets whose destination port did not match the current listening port. This behavior has been modified. The UDP destination port field is subtracted from the base listening port. If the result is 0 then the packet is aimed at the first port; else, if the result is 1 the packet is aimed at the second port; otherwise the packet is dropped.

**CWDP RX** The CWDP RX block is used for both ports and it must keep track of the packet sequence numbers for the two ports. A block diagram is depicted in Figure 3.9. The control signal `CWLinkPort` produced by the UDP RX block identifies the currently active port, and is used to select the packet sequence number register to check and increment. The `CWLinkPort` signal is also decoded to produce a 2-bit signal (`active_unit_RX`) to select the active CW-Link RX block. Upon reception of a `SET_CORE_ACCESS` packet from a given UDP port, one of two registers named (`locked_UDP_0` and `locked_UDP_1`) is set with the source UDP port. These are provided to the CWDP TX block so it can choose the correct destination UDP port to reply to. See Figure 3.11 for details.

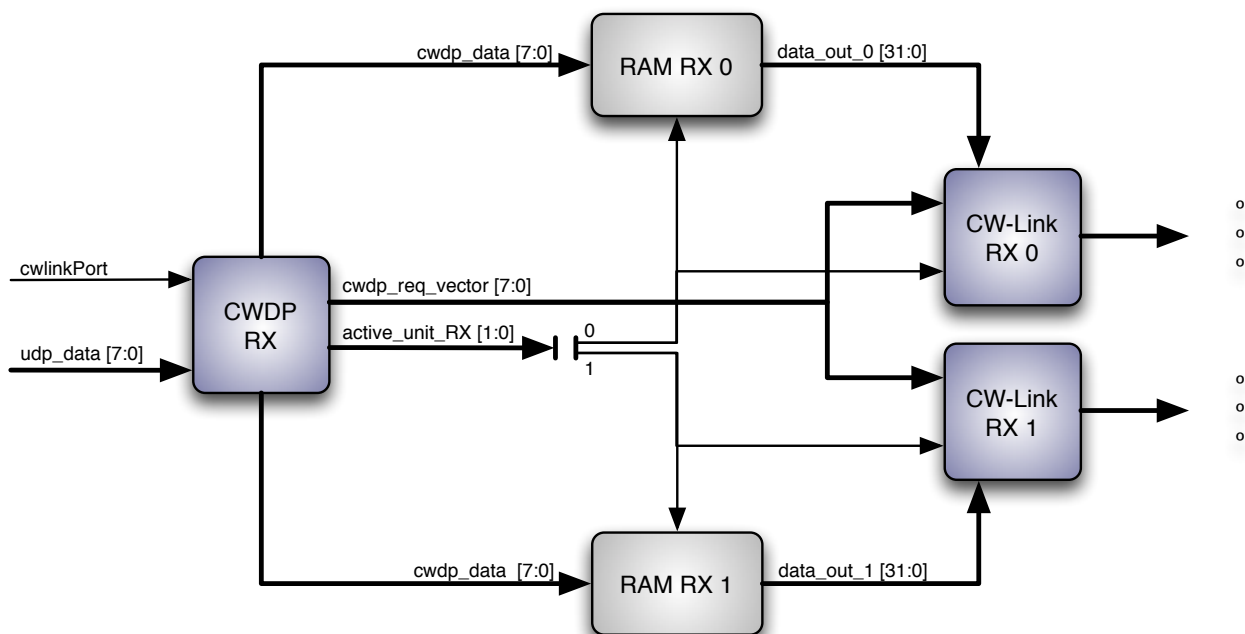


Figure 3.9: CWDP RX and CW-Link RX



**CW-Link RX** The CW-Link RX unit is duplicated, one for each communication port. Its control Finite State Machine (FSM) has been modified so that they are activated one at a time, based on the value of the control signal `active_unit_RX`.

**CW-Link TX** The CW-Link TX unit is duplicated. Its control FSM has been modified to allow execution only when the Arbiter block grants access to the (RAM TX and CWDP TX) resources. To do this, two signals are sent to the arbiter, `cw_req_vector_TX` and `busy_cwlink`, and one signal is received from the arbiter, `active_unit_tx`, as shown in Figure 3.10. When the resources are granted to one of the CW-Link TX units the arbiter waits until its execution ends before granting the resources to the other CW-Link TX unit.

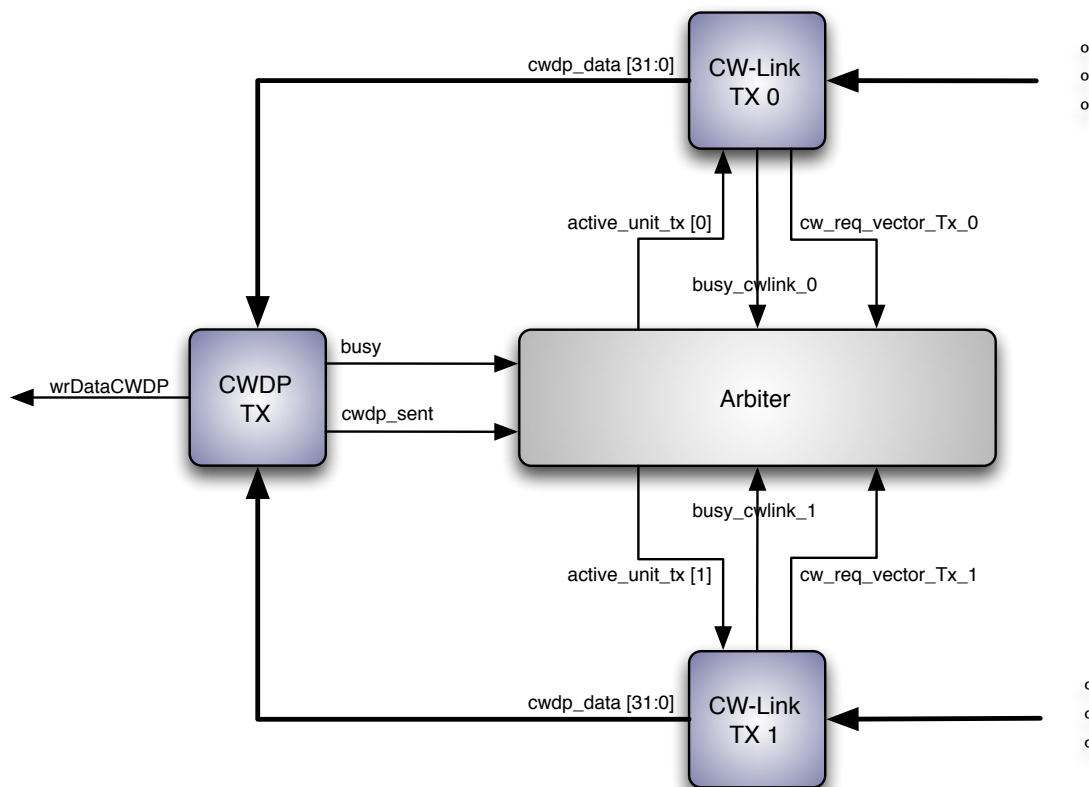


Figure 3.10: CWDP TX and CW-Link TX

**CWDP TX** A single CWDP TX block is used. The UDP destination port may be different depending on the type of packet: ACK packet requested by the CWDP RX block or data packet requested by one of the two CW-Link TX units, as shown in Figure 3.11. If the CWDP TX is processing a CORE\_DATA packet, the destination UDP port is selected from the `locked_UDP_0` or `locked_UDP_1` inputs, based on the currently active CW-Link TX block. For an ACK packet request, the `SendACK`

### 3. Design

---

signal is activated, the `Se1ACK` signal selects the UDP destination port from the `locked_UDP_0` or `locked_UDP_1` inputs, and the sequence number is provided by the `SeqNumACK` signal.

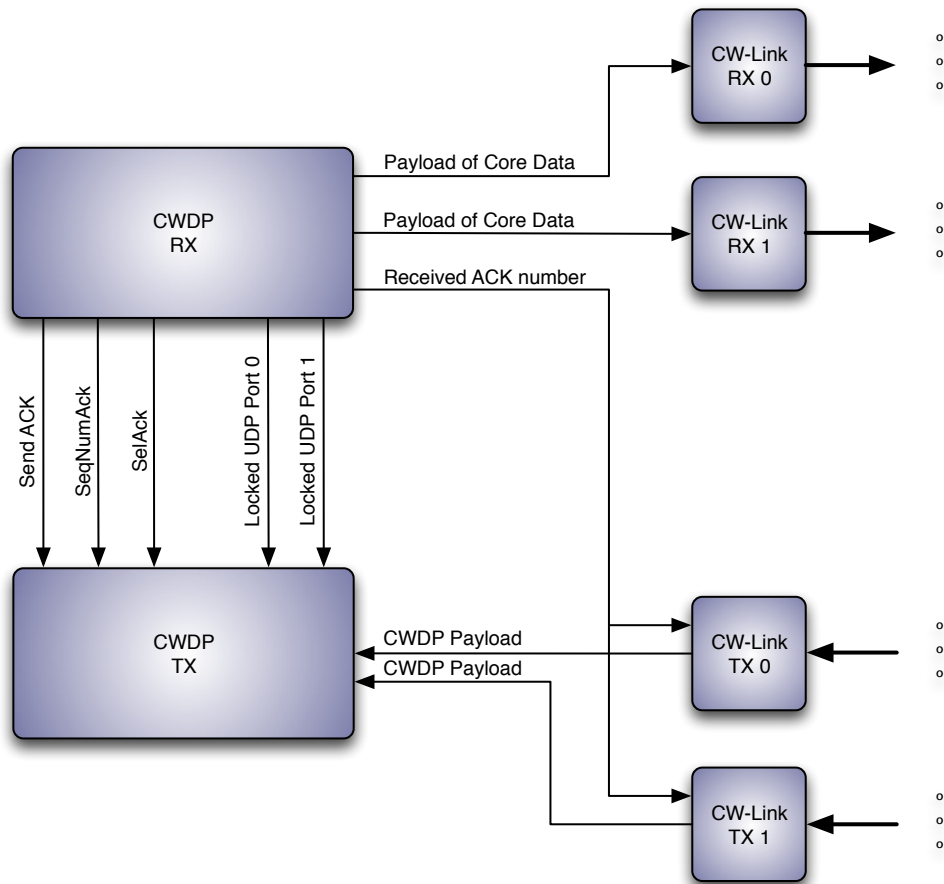


Figure 3.11: CWDP Layer

### 3.2.2 Arbiter

To control the access to both the CWDP TX and the RAM TX modules from the CW-Link TX modules an arbiter block has been implemented. The arbiter FSM is described in Table 3.1 (states), Table 3.2 (input/output signals), and Figure 3.12 (state transition diagram). The arbiter's algorithm is simple and tries to ensure that both UDP ports get the same priority in using Face-Works' transmission infrastructure: when both ports want to have access to the CWDP layer, control is toggled from the port that currently has control to the port that does not.

Table 3.1: Arbiter FSM States

State	Description
s_stop	Initial state. FSM waits for a TX request from one of the Cw-Link units.
s_stop_1	FSM waits for a TX request from one of the Cw-Link units.
s_tx_0	Resources Granted to Cw-Link 0.
s_tx_1	Resources Granted to Cw-Link 1.
s_wait_0	Waiting for Cw-Link 0 to free the Resources.
s_wait_1	Waiting for Cw-Link 1 to free the Resources.

Table 3.2: Arbiter Input and Output Signals

Signals	Direction	Description
CWDP		
busy	input	Signals that the CWDP TX unit is being used.
cdwp_sent	input	Signals that the packet was sent.
CW-Link 0		
active_unit_tx[0]	output	Activates the CW-Link 0 unit.
busy_cwlink_0	input	Signals that the CW-Link 0 is busy.
cw_req_vector_TX_0	input	Signals that the Cw-Link 0 is requesting access.
CW-Link 1		
active_unit_tx[1]	output	Activates the CW-Link 1 unit.
busy_cwlink_1	input	Signals that the CW-Link 1 is busy.
cw_req_vector_TX_1	input	Signals that the Cw-Link 1 is requesting access.

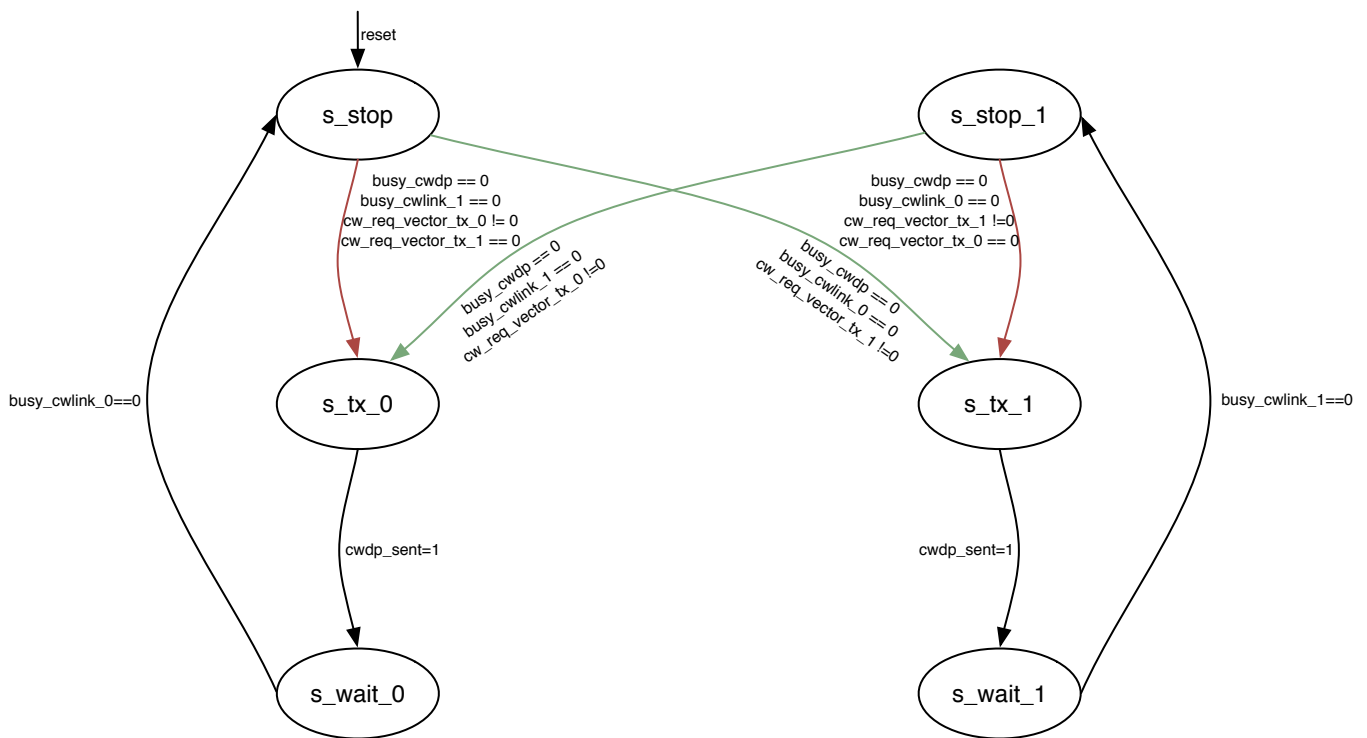


Figure 3.12: Arbiter FSM State Transition Diagram

# 4

## Verification

### Contents

---

4.1 Verilator . . . . .	38
4.2 Verification Environment . . . . .	39
4.3 Verilator Module Instantiation . . . . .	43
4.4 Running the Simulation Model . . . . .	46

---

## 4. Verification

---

In this chapter the methodology used to debug, verify and inspect the FaceWorks core is presented.

Initially, verification of the design was being performed directly on the FPGA using as main tools for debug a packet sniffer (Wireshark), an Integrated logic analyzer (ILA) (Xilinx's ChipScope Pro) and the Face-Test application to send stimuli to the FPGA board and analyze the responses. However, as the design got near completion, some intermittent bugs showed up, which were very difficult to identify using this ad-hoc verification method.

The main difficulty was to know when and what signal was the causing a problem, so that correct trigger conditions could be set. Using the ILA proved ineffective due to too many inconclusive trigger conditions, and a limited buffer size to store data samples for analysis, which is the result of having limited number of Block Random Access Memorys (BRAMs) in the device.

An alternative consisted in creating a testbench and running the system into a simulator such Xilinx's iSim. However, it was difficult to recreate realistic test conditions, unless a significant amount of C and Verilog code were developed. Another way is to modify the system in order to be able to run a simpler to write testbench, but this approach is risky as important features may end up untested.

Besides, methods for the testbench to read and write data are limited to file Input-Output (IO) since no Verilog Procedural Interface (VPI) support is offered for iSim. Writing and reading files to exercise the MII interface in a testbench is hard and complicated to implement, considering the sheer amount of data needed to adequately test a network interface. So other alternatives were searched so that the design could be properly debugged and verified. Of the considered alternatives, Verilator, a free open source application, has been chosen to perform this task.

This chapter describes Verilator, explaining C++ testbenches, the process of instantiating Verilator generated SystemC models in C++ applications, and how to run simulations.

### 4.1 Verilator

Verilator is an application that translates a synthesizable Verilog description into an optimized cycle-accurate behavioral model in C++/SystemC, which is called a verilated file. Verilator is a two state simulator (0,1), but it has features to deal with unknown states. The testbench is a C/C++ application that wraps the verilated model and is compiled with it using a common C++ compiler. Verilated models show high simulation speed, which is on par or higher than commercial simulators such as NC-Verilog, VCS and others. [18].

#### 4.1.1 Verilator History

Verilator was originally created in 1994 by Paul Wasson at the Core Logic Group at Digital Equipment Corporation (DEC). In 1998 DEC released the source of verilator under a GNU Public

License. The maintainer of the project since 2001, Wilson Snyder, added SystemC support and fully rewrote Verilator in C++. Latest additions provide SystemVerilog and SystemVerilog Direct Programming Interface (DPI) language support[19, p. 65].

#### 4.1.2 Verilator Testbench

Using Verilator, a FaceWorks simulator has been created which can be stimulated with the same socket-based test application used for exercising FaceWorks in an FPGA. In this way test patterns that fail in the FPGA device can be replicated and analyzed in simulation with full visibility over all design signals. A FaceWorks SystemC model is created by running Verilator on the FaceWorks Verilog code, and this object is then instantiated in the testbench application.

## 4.2 Verification Environment

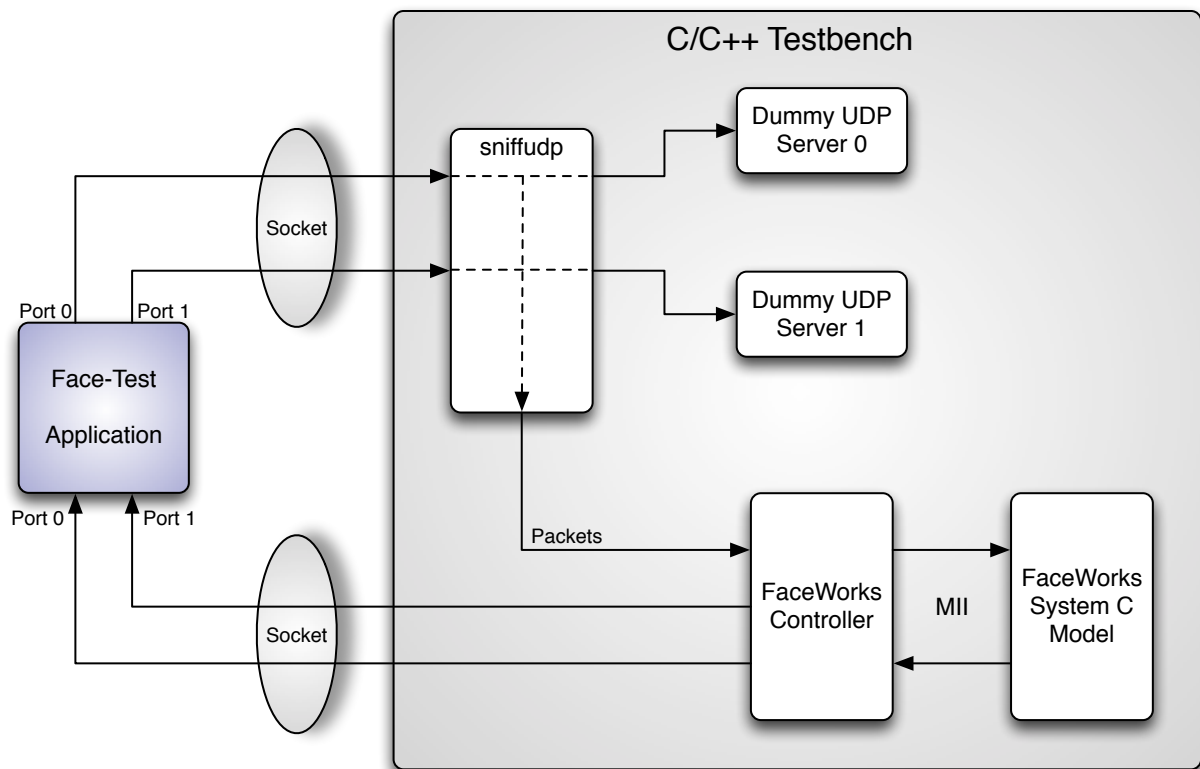


Figure 4.1: Verification Environment

Figure 4.1 shows the verification environment for FaceWorks, encompassing the Face-test application and the C++ testbench. The testbench has five components: 3 threads (Dummy UDP Server 0, Dummy UDP Server 1 and sniffudp), the FaceWorks Controller and the FaceWorks model.

## 4. Verification

---

The two threads `Dummy UDP Server 0` and `Dummy UDP Server 1` perform the simple operation of reading the UDP ports persistently, so the input data coming from the application do not fill the OS socket buffer.

The `sniffudp` thread launches the data capture code, which is based on the `sniffex.c` code example of `libcap`, a portable C/C++ library for network traffic capture [20].

This data capture code must be used instead of a simple UDP socket, because this way the full Ethernet frame is captured, which makes it easier to reconstruct the MII packet in order to feed the data to the MII interface of the FaceWorks model.

The FaceWorks Controller is responsible for receiving data from the `sniffudp` thread, formatting the data according to the MII format and deliver them to the FaceWorks MII interface. The FaceWorks Controller is also responsible for unpacking the data it receives from the FaceWorks model in order to send them to the test application via UDP sockets.

The FaceWorks model is a SystemC Model created by running Verilator on the FaceWorks Verilog code.

Threads are used so that the FaceWorks Controller can run at the same time as data is intercepted by the `sniffudp` function. These two components are explained in detail in the next two subsections.

### 4.2.1 sniffudp

A simpler solution to capture data from the Face-Test application is to implement a C/C++ UDP socket server in the testbench to read the incoming data from the Face-Test application and provide it to the FaceWorks SystemC model. However, since the FaceWorks connection to the outside is the MII interface, one would need to recreate the MII packet from the received UDP payload. Recreating the MII packet implies recreating the UDP packet followed by the IP packet, followed the Ethernet frame. This would be unnecessarily complicated.

A solution for this problem is to capture the packets at the data link layer, which can be accomplish by the use of `libpcap`, avoiding the process of recreating the Ethernet, IP and UDP packets.

The `sniffudp` function is set to capture data on the Linux loopback (`lo`) device, which are addressed with the two FaceWorks UDP ports. The `lo` device is a virtual network interface used to receive packets sent from the host to itself. After the setup the `sniffudp` thread blocks waiting for filtered packets. When such packets arrive they are passed to a callback function `got_packet()`.

The `got_packet()` function receives almost complete Ethernet frames (Figure 2.4) which miss the FCS field and have a blank destination and source address, since the data is traveling through the loopback interface. The fields preamble, SFD, destination address, source address and Ether-type are inserted. The data field (containing the IP packet) is copied from the sniffed packet. The CRC is calculated as described in section 4.2.4 and appended as the FCS field.

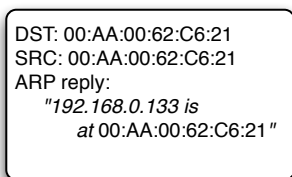


Then, the `got_packet()` function reaches a critical region where it pushes the packet into a queue shared with the main thread, after which it terminates execution. When another packet is filtered the callback function is called again and the process repeats all over again.

### 4.2.2 FaceWorks Controller

The FaceWorks Controller instantiates the FaceWorks model, and communicates with it. The communication with the model is presented in this section, and the instantiation of the model is presented in 4.3.

Since the FaceWorks Controller does not implement the ARP protocol, to properly support the communication between the Face-Test application and the FaceWorks model, the FaceWorks controller starts by sending a fixed ARP reply packet as shown in Figure 4.2. This packet is sent even without receiving any ARP request from the FaceWorks model. The purpose is to fill the ARP cache with an IP address / MAC address pair, so the FaceWorks Ethernet TX module can obtain the MAC address by consulting the ARP module. If the cache is already stuffed, no ARP query packets will be generated by the FaceWorks model.



```
DST: 00:AA:00:62:C6:21
SRC: 00:AA:00:62:C6:21
ARP reply:
  "192.168.0.133 is
    at 00:AA:00:62:C6:21"
```

Figure 4.2: Injected ARP Reply Packet

After the ARP stuffing is completed, the FaceWorks controller is in a loop, trying to transfer data from the packet queue, shared with the `sniffudp` thread, to the MII RX interface of the FaceWorks model, and/or to transfer data from the MII TX interface to the socket that leads to the Face-Test application.

To transfer data from the queue to the MII RX interface the FaceWorks controller needs to get hold of the mutex that protects the shared packet queue. When access to the mutex is granted and at least one packet exists in the queue, the packet is popped from the queue and fed into the MII RX interface of the FaceWorks model, nibble by nibble.

To transfer data to the socket leading to Face-Test, the FaceWorks controller stores the received nibbles in a buffer, which is then passed to another function, the `process_recv_packet` function discussed in the next section, which verifies and forwards the data to the Face-Test application.

## 4. Verification

---

### 4.2.3 process\_rcv\_packet

The `process_rcv_packet` function receives as argument a packet buffer from the FaceWorks controller, processes the packet and sends it over the network socket to the Face-Test application.

The `process_rcv_packet` function checks if the Preamble, SFD and Ethernet headers are correctly constructed, by comparing their values to the expected values. The FCS field is calculated for the received packet and compared with the FCS value in the packet itself. The FCS field contains a CRC checksum computed as described in section 4.2.4. If any field does not match the expected value, the function returns with an error which indicates the malformed field.

If a well formed packet is received, the IP headers and UDP headers are read to extract the destination IP address, UDP port and UDP payload offset, and the UDP packet is sent to the Face-Test Application.

By verifying the data from the MII interface, the operation of the FaceWorks model is always under scrutiny, which is a valuable debug feature. Next, the Ethernet, IP and UDP headers are removed, and the data is placed into the UDP socket to be sent to the Face-Test application, which is listening to the UDP ports in use.

### 4.2.4 Cyclic Redundancy Check

The CRC is computed by both the `process_rcv_packet` and the `got_packet` function. The source code used to calculate the CRC has been generated by a Python script denominated `pycrc`[21]. The `pycrc` script outputs a C header file (.h) and C source code file (.c) which can compute the CRC.

```
> python pycrc.py --model crc-32 --algorithm table-driven --poly 0x04c11db7 --generate h -o crc.h
> python pycrc.py --model crc-32 --algorithm table-driven --poly 0x04c11db7 --generate c -o crc.c
```

Figure 4.3: Generating CRC Functions with the `pycrc` Python Script

Figure 4.3 illustrates how to use the `pycrc` script. The script must be called twice: the first time to generate the .h file and the second time to generate the .c file. The command line arguments provided are the appropriate ones to generate the Ethernet FCS, a 32-bit CRC which uses the 0x04c11db7 polynomial. The `--algorithm table-driven` option has been included because it produces the fastest algorithm. This method uses a look-up table which might not be acceptable for embedded systems. But, since this is only used in the testbench program which runs on the PC, memory restrictions do not apply.

## 4.3 Verilator Module Instantiation

The Verilog module declaration of the FaceWorks system is depicted in Figure 4.4. The testbench code has been written based on the `test_sc` and `test_sp` examples included as part of the Verilator installation directory `verilator/examples/` [19].

```
module faw_system(
    input wire sys_clk,
    input wire sys_rst,
    ///IP/PORT CONFIG PINS
    input wire[3:0] faw_dyn_conf,
    input wire faw_sub_net,

    ///ETHERNET PHY PINS
    input wire MAC_PHY_tx_clk_pin;
    input wire MAC_PHY_rx_clk_pin;
    input wire MAC_PHY_rx_dv_pin;
    input wire [3:0] MAC_PHY_rx_data_pin;
    output wire MAC_PHY_tx_en_pin;
    output wire MAC_PHY_tx_error_pin;
    output wire [3:0] MAC_PHY_tx_data_pin;
    output wire MAC_PHY_rst_n;
    output wire MAC_PHY_mdc;
);
```

Figure 4.4: FaceWorks Verilog Module Declaration (`faw.system.v`)

In the testbench code quite a few declarations are needed to instantiate the FaceWorks verilated model. For this it is necessary to include the following header files, as shown in Figure 4.5.

```
#ifndef SYSTEMPERL
# include "systemperl.h" // SystemC + SystemPerl global header
# include "sp_log.h" // Logging cout to files
# include "SpTraceVcd.h"
# include "SpCoverage.h"
#else
# include "systemc.h" // SystemC global header
# include "verilated_vcd_sc.h" // Tracing
#endif

#include "Vfaw_system.h" // Top level header, generated from verilator
```

Figure 4.5: Testbench Header Files

The testbench code supports two type of simulations: the SystemC simulation (signal tracing) and the SystemC + SystemPerl simulation (code coverage and signal tracing). The `#ifdef` clause in Figure 4.5 selects the correct header files depending on the simulation case. Then the header file `Vfaw_system.h`, which results from running Verilator run on the FaceWorks Verilog code, is included.

The testbench is written in SystemC and therefore the main function is denoted `sc_main`, as shown in Figure 4.6.

## 4. Verification

```
int sc_main(int argc, char* argv[]) {
    (...)
    //
    // Define the Clocks
    //
    #if (SYSTEMC_VERSION>=20070314)
    sc_clock sys_clk      ("sys_clk", 20,SC_NS, 0.5,10,SC_NS, false);
    sc_clock MAC_PHY_tx_clk_pin ("MAC_PHY_tx_clk_pin", 40, SC_NS, 0.5, 0, SC_NS, false);
    sc_clock MAC_PHY_rx_clk_pin ("MAC_PHY_rx_clk_pin", 40, SC_NS, 0.5, 0, SC_NS, false);
    #else
    sc_clock sys_clk      ("sys_clk", 20, 0.5, 10, false);
    sc_clock MAC_PHY_tx_clk_pin ("MAC_PHY_tx_clk_pin", 40, 0.5, 0, false);
    sc_clock MAC_PHY_rx_clk_pin ("MAC_PHY_rx_clk_pin", 40, 0.5, 0, false);
    #endif

    //=====
    // Define the Interconnect
    cout << "Defining Interconnect\n";
    sc_signal<bool> sys_rst;
    sc_signal<vluint32_t> faw_dyn_conf;
    sc_signal<bool> faw_sub_net;
    sc_signal<bool> MAC_PHY_tx_en_pin;
    sc_signal<bool> MAC_PHY_tx_error_pin;
    sc_signal<vluint32_t> MAC_PHY_tx_data_pin;
    sc_signal<bool> MAC_PHY_rx_dv_pin;
    sc_signal<vluint32_t> MAC_PHY_rx_data_pin;
    sc_signal<bool> MAC_PHY_rst_n;
    sc_signal<bool> MAC_PHY_mdc;
```

Figure 4.6: Signal Declaration in the SystemC Testbench

To declare the SystemC clock and signal objects, the `sc_clock` and `sc_signal` classes are used. The `sys_clk` object is a clock with a period of 20 ns (SC\_NS is used to specify time units), a duty cycle of 50%, the first transition occurs at 10 ns, and is a negative edge transition (`false`). This signal drives the FaceWorks clock pin. The other two clock signals, (`MAC_PHY_tx_clk_pin` and `MAC_PHY_rx_clk_pin`), are used in the MII/MAC interface, and have similar declarations.

For the rest of the signals (ie, signals that are not clocks), the `sc_signal` class is used, which declares signals of various data types. Single bit signals use the `bool` type, 2 to 32-bit vectors use the `vluint32_t` type, 33 to 64-bit vectors use the `vluint64_t` or the `sc_bv` types<sup>1</sup>, and wider bit vectors use the `sc_bv` type. When Verilator translates Verilog code to SystemC, the SC\_MODULE pinout will show the type conversions described, as can be seen in the output header file `Vfaw_system.h`, where only `bool` and `vluint32_t` types are used. The reasons for these type conversions are linked with the simulator's performance.

<sup>1</sup>The `sc_bv` is also used for 33 to 64-bit vectors if the `-no-pins64` flag is provided to Verilator

```
//=====
// Device under test
#ifdef SYSTEMPERL
SP_CELL (top, Vfaw_system);
SP_PIN (top, sys_clk,sys_clk);
SP_PIN (top, MAC_PHY_tx_clk_pin,MAC_PHY_tx_clk_pin);
SP_PIN (top, MAC_PHY_rx_clk_pin,MAC_PHY_rx_clk_pin);
SP_PIN (top, sys_rst,sys_rst);
SP_PIN (top, faw_dyn_conf,faw_dyn_conf);
SP_PIN (top, faw_sub_net,faw_sub_net);
SP_PIN (top, MAC_PHY_tx_en_pin,MAC_PHY_tx_en_pin);
SP_PIN (top, MAC_PHY_tx_error_pin,MAC_PHY_tx_error_pin);
SP_PIN (top, MAC_PHY_tx_data_pin,MAC_PHY_tx_data_pin);
SP_PIN (top, MAC_PHY_rx_dv_pin,MAC_PHY_rx_dv_pin);
SP_PIN (top, MAC_PHY_rx_data_pin,MAC_PHY_rx_data_pin);
SP_PIN (top, MAC_PHY_rst_n,MAC_PHY_rst_n);
SP_PIN (top, MAC_PHY_mdc,MAC_PHY_mdc);
#else
Vfaw_system* top = new Vfaw_system("top");
top->sys_clk (sys_clk);
top->MAC_PHY_tx_clk_pin (MAC_PHY_tx_clk_pin);
top->MAC_PHY_rx_clk_pin (MAC_PHY_rx_clk_pin);
top->sys_rst (sys_rst);
top->faw_dyn_conf (faw_dyn_conf);
top->faw_sub_net (faw_sub_net);
top->MAC_PHY_tx_en_pin (MAC_PHY_tx_en_pin);
top->MAC_PHY_tx_error_pin (MAC_PHY_tx_error_pin);
top->MAC_PHY_tx_data_pin (MAC_PHY_tx_data_pin);
top->MAC_PHY_rx_dv_pin (MAC_PHY_rx_dv_pin);
top->MAC_PHY_rx_data_pin (MAC_PHY_rx_data_pin);
top->MAC_PHY_rst_n (MAC_PHY_rst_n);
top->MAC_PHY_mdc (MAC_PHY_mdc);
#endif
#endif
```

Figure 4.7: Interconnecting the Verilated Model and the Testbench Signals

After all signals are declared, they are used to connect the testbench signals to the Verilator module pins. Another `#ifdef` clause is used to select if the verilated module is instantiated using SystemPerl or SystemC, as show in Figure 4.7. With the SystemC model instantiated and the signals connected in the C++ testbench, the model is ready to be initialized and run.

### 4.4 Running the Simulation Model

To ease the compiling and running of the simulation model, two different Makefile sets have been created. One set of makefiles generates the SystemC model and runs the simulation to produce wave traces. The other set of Makefiles creates the SystemC and SystemPerl code to produce wave traces and coverage metrics.

#### 4.4.1 Folder Structure

In Figure 4.8, a representation of the folder structure used for the project is shown. Each folder is explained as follows:

- `faw_sys_verilog`: contains the Verilog source of FaceWorks;
- `test_sc`: contains the Makefiles and the testbench source code for generating the SystemC simulation;
- `test_sp`: contains the makefiles for the SystemC/SystemPerl simulation;
- `src`: contains files that are used as input to verilator.

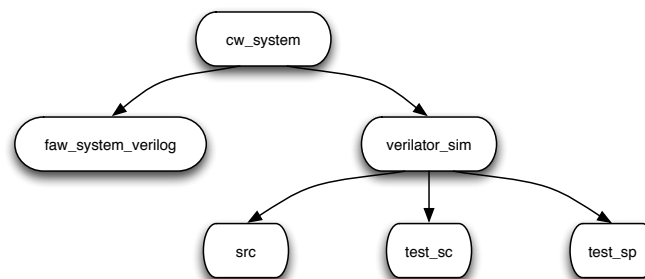


Figure 4.8: Project Folder Hierarchy

#### 4.4.2 Producing the Simulation Model

The SystemC simulation model is produced by the Makefile in the `test_sc` folder, which starts by invoking Verilator, as shown in Figure 4.9, with the following flags:

- `-sc`: generates SystemC output;
- `-f`: specifies an input file with commands;
- `-trace`: produces waveform traces while running.

The Verilator input file `input.vc` placed in the `src` folder is shown in Figure 4.10. The `input.vc` file contains directives for file extensions and directories where the Verilog sources can be found.

```
verilator --sc -f ../src/input.vc faw_system.v --trace
```

Figure 4.9: Verilator Invocation

```
+librescan +libext+.v  
+incdir+../../faw_sys_verilog
```

Figure 4.10: Verilator Directives

After the input files are verilated the Makefile will call the Makefile\_obj file to compile the test-bench source code and link it with the verilated model.

The resulting application needs to capture network packets without running in super-user mode. This is accomplished by specific Linux settings and by using the `setcap` application. The makefile automatically makes these settings and runs the simulator application.

### 4.4.3 Linting the Verilog Code

When executing, Verilator first verifies if lint violations exist, to ensure the code is synthesizable. If any warning or error exists in the code Verilator aborts execution unless the particular line of code causing the error or warning is protected with a pair of pragmas, as depicted in Figure 4.11.

```
/*verilator lint_off PINNOCONNECT*/  
.crcValid(/* open */),  
/*verilator lint_on PINNOCONNECT*/
```

Figure 4.11: A Linting Warning Disabled

Warnings can be ignored by calling verilator with the flag `-Wno-warning`. Verilator may also be used exclusively as a "linting" tool, invoking Verilator with the flag `-lint-only`. Some of the warnings/errors produced by Verilator can have an impact on the performance of the model and produce non-synthesizable code; so it is recommended to correct all linting warnings before proceeding. Some examples of the verilator warnings that occurred while processing FaceWorks are illustrated below. For each one of these warnings the problem was identified and solved.

## 4. Verification

---

In Figure 4.12, the Case Overlap and Case Incomplete warnings produced by a case statement were due to a user error in the Verilog description. One of the case values overlaps the case value (0x4), and leaves the case value 0x07 uncovered, as no default case statement is given.

```
%Warning-CASEOVERLAP: ../../faw_sys_verilog/faw_cwdp_rx.v:571:
Case values overlap (example pattern 0x4)
%Warning-CASEOVERLAP:
Use "/* verilator lint_off CASEOVERLAP */" and lint_on around source to disable this message.
%Warning-CASEINCOMPLETE: ../../faw_sys_verilog/faw_cwdp_rx.v:555:
Case values incompletely covered (example pattern 0x7)
```

Figure 4.12: Case Overlap and Case Incomplete Warnings

This specific situation, which was caused by a typo, is easy to correct by changing the value of the overlapping statement to the expected value, fixing both warnings. In other cases, where a full coverage of input signal combinations is not obtained, one needs to insert the default statement, so that the warning does not display.

In Figure 4.13, the 11-bit signal CWDp\_data\_size is added to the 4-bit constant in the Right Hand Side (RHS) of the expression. Although this is a valid Verilog expression, and accepted by tools such as Xilinx Synthesis Tool (XST), this code produces the Verilator warning depicted in Figure 4.14.

```
packet_size_next = CWDp_data_size + 4'h 4;
```

Figure 4.13: Addition with Narrower RHS Warning

```
%Warning-WIDTH: ../../faw_sys_verilog/faw_cwdp_tx.v:255:
Operator ADD expects 11 bits on the RHS, but RHS's CONST '4'h4' generates 4 bits.
%Warning-WIDTH:
Use "/* verilator lint_off WIDTH */" and lint_on around source to disable this message.
```

Figure 4.14: Width Mismatch Warning

To solve the Width Mismatch warning, the missing Most Significant Bits (MSBs) on the RHS of the expression must be declared as depicted in Figure 4.15. This way the two operands and the sum are 11-bit vectors. A 12-bit vector for the sum would also be acceptable.

```
packet_size_next = CWDp_data_size + {7'b0,4'h 4};
```

Figure 4.15: Correction of the Previous Width Mismatch Problem



The warning presented in Figure 4.16 identifies a path that contains circular logic. It is unlikely that the resulting asynchronous logic will meet the desired functionality. Also, this may affect the Verilator model performance, so it is recommended to fix this problem.

```
%Warning-UNOPTFLAT: ../../faw_sys_verilog/faw_faceworks.v:302:
Signal unoptimizable: Feedback to clock or circular logic: v.facework.active_unit_tx
%Warning-UNOPTFLAT: Example path: ../../faw_sys_verilog/faw_faceworks.v:302:  v.facework.active_unit_tx
%Warning-UNOPTFLAT: Example path: ../../faw_sys_verilog/faw_faceworks.v:749:  ASSIGNW
%Warning-UNOPTFLAT: Example path: ../../faw_sys_verilog/faw_faceworks.v:270:  v.facework.CWDP_nr
%Warning-UNOPTFLAT: Example path: ../../faw_sys_verilog/faw_cwdp_tx.v:175:  ALWAYS
%Warning-UNOPTFLAT: Example path: ../../faw_sys_verilog/faw_faceworks.v:357:  v.facework.busy_cwdp
%Warning-UNOPTFLAT: Example path: ../../faw_sys_verilog/faw_faceworks.v:681:  ALWAYS
%Warning-UNOPTFLAT: Example path: ../../faw_sys_verilog/faw_faceworks.v:302:  v.facework.active_unit_tx
%Error: Exiting due to 2 warning(s)
%Error: Command Failed /usr/local/bin/verilator_bin --sc -f ../src/input.vc faw_system.v --trace
```

Figure 4.16: Circular Logic Warning

As one can see in Figure 4.16, the combinatorial loop starts and ends in the signal `active_unit_tx`. This warning is corrected by changing the assignments done to the `active_unit_tx` signal within the FSM to break the existing feedback loop.

### 4.4.4 Observing Wave Traces

Assuming no warning or errors were detected during the lint and compile phases, the simulator model is run and the Face-Test application stimulates it. After a predefined number of exchanged packets, the simulator terminates execution and dumps a VCD file that contains the simulation waveforms.

To display the contents of the VCD file, the Makefile launches the GTKwave application [22]. With GTKWave the full hierarchy of the design is shown, and the desired signals or registers can be added to the visualization window. In Appendix A, a screen shot showing the GTKwave application can be seen.

### 4.4.5 Coverage

To generate coverage metrics the SystemPerl Makefile inside the `test_sp` folder must be used. This causes a coverage file name `coverage.pl` to be written in the `verilator_sim/test_sp/logs/` folder when the simulation ends. Naturally, coverage results depend on the quality of the stimulus provided by the Face-Test application.

## 4. Verification

---

The application Vcoverage, part of the SystemPerl toolkit, can output a copy of the source code files of the project with coverage metrics annotated. By default, the file `logs/coverage.pl` is read to annotate the source code and place a copy under `logs/coverage_source/`, as shown in Figure 4.17.

```
> vcoverage
Total coverage (398/687) 57.93%
See lines with '%00' in logs/coverage\_source
```

Figure 4.17: Output from Vcoverage

For different stimulus patterns, distinct output coverage files are obtained. To combine these into a single coverage file the `vcoverage` application is called with the arguments presented in Figure 4.18.

```
> vcoverage --noreport -write logs/merged-cov.dat *.pl
Generating the report
> vcoverage --min 1 --o logs/coverage_source logs/merged-cov.dat
Total coverage (524/687) 76.27%
See lines with '%00' in logs/coverage\_source
```

Figure 4.18: Merging Distinct Coverage Files and Generating Output Statistics

First, all `.pl` files are merged into a single `merged-cov.dat` file by using the `--noreport` option. Then, coverage statistics are generated for the joined files and the output source code files are placed in the same output directory `logs/coverage_source`.

```
014988 s_wait_1 : begin
000026     if((busy_cwlink_1 == 1'b 0)) begin
        // Ack foi recebido
        active_unit_tx_next=2'b 00;
        next_state = s_stop_1;
    end
end

%000000 default: begin
    active_unit_tx_next = 2'b 00;
    next_state = s_stop;
end
```

Figure 4.19: Code Coverage Report Excerpt

As shown in the Verilog code example in Figure 4.19, the number of times each code block is executed is annotated on the left. Code that `vcoverage` thinks needs more coverage is marked % X, where X is the number of times the line has been exercised. Any line exercised less than a minimum number of times (which can be specified with the `--min` option) is marked % X.

# 5

## Results

### Contents

---

5.1	FPGA Implementation Results . . . . .	52
5.2	Ethernet Switch Characterization . . . . .	53
5.3	Throughput Upper Bound Model . . . . .	55
5.4	FaceWorks Sustained Throughput . . . . .	56

---

## 5. Results

In this chapter two kind of results are presented: 1) FPGA implementation results on consumed resources and operation frequency; 2) bandwidth results.

### 5.1 FPGA Implementation Results

The system has been developed and tested on a Xilinx SP605 board, featuring a Spartan 6 XC6SLX45T device and a Marvell Alaska PHY (88E1111) chip. The Spartan 6 is a low end device, designed for larger production volumes and low price but with limited performance.

The synthesis results presented are obtained with the Xilinx ISE 14.4 suite of tools. In Table 5.1, synthesis results for the original single port implementation are presented. Table 5.2 presents synthesis results for the two-port implementation designed in this work.

Table 5.1: Implementation Results for the Single-Port FaceWorks

	Used	Total	Percentage Used
Slice Logic Utilization			
Number of Slice Registers	1527	54576	2%
Number of Slice LUTs:	2178	27288	7%
Number used as Logic	2036	27288	7 %
Number used as Memory	16	6048	1 %
Slice Logic Distribution			
Number of LUT Flip Flop pairs used:	2375	-	-
Number with an unused Flip Flop	999	2375	42%
Number with an unused LUT	197	2375	8 %
Number of fully used LUT-FF pairs	1179	2375	49 %
Specific Feature Utilization			
Number of Block RAM (16Kb)	2	116	2%

Table 5.2: Implementation Results for the Two-Port FaceWorks

	Used	Total	Percentage Used
Slice Logic Utilization			
Number of Slice Registers	1917	54576	3%
Number of Slice LUTs:	2965	27288	10%
Number used as Logic	2933	27288	10 %
Number used as Memory	32	6048	1 %
Slice Logic Distribution			
Number of LUT Flip Flop pairs used:	3963	-	-
Number with an unused Flip Flop	2046	3963	51%
Number with an unused LUT	998	3963	25 %
Number of fully used LUT-FF pairs	919	3963	23 %
Specific Feature Utilization			
Number of Block RAM (16Kb)	3	116	2%

The synthesis results show that both FaceWorks designs are very economic in terms of size. The single-port design uses the equivalent of 14000 gates, while the two-port design uses about 20000 gates<sup>1</sup>. However, it must be noted that the single-port design only supports 8 CW-Link connections, while the two-port design supports 16 CW-Link connections. The IP cores use very little memory and zero DSP blocks. The overall size can be considered adequate for a test and debug core. For a small FPGA like the Spartan 6, it only occupies 10 % of the logic resources in the two-port configuration.

The results show that with about 7000 gates a new UDP port and 8 new CW-Link connections can be added to the design. Thus, the design scales almost linearly, being the non-linear part the small amount of logic needed to augment the size of the arbiter.

Table 5.3: Period Timing Constraints

	Used	Minimum
Sys_clk	20 ns	8.341 ns
TX_CLK	40 ns	NA
RX_CLK	40 ns	NA

The system clock period used and the minimum period constraint that has been possible to meet are shown in Table 5.3. The minimum period (8.341 ns), which corresponds to about 120MHz is a competitive frequency for a Spartan 6 device. Compared to the single port implementation, where the minimum clock period is 6.912 ns, the minimum clock period increases 20% for the two-port implementation. This is mostly due the Arbiter block design, which scales poorly in terms of frequency. A better design would not be very difficult to implement but falls out of the scope of this thesis. For a number of ports higher than two, it is highly recommended that the Arbiter design is reviewed.

## 5.2 Ethernet Switch Characterization

The results presented in this thesis have been obtained on a notebook with an Intel U4100 @ 1.30GHz processor, 3 GB of RAM and an Atheros AR8131 ethernet adapter, running the Linux kernel 3.2.0-x86-64. To interconnect Ethernet devices a TP-Link TL-WR841N switch has been used. Two network tests have been performed to determine the switch capabilities, and detect if in either case they could limit the FaceWorks performance.

The first test was designed to determine the maximum throughput achievable with the switch. This test consisted in exchanging data in both directions, simultaneously, between two notebooks interconnected via the switch. To establish the network connections between the notebooks the `netcat (nc)` application was used in UDP mode. To avoid limitations in hard disk throughput, the outputs were redirected to the null device (`/dev/null`).

<sup>1</sup>This figure has been obtained by counting 6 gates per each 6-input LUT and one gate per flip-flop

## 5. Results

---

The throughput values presented in Table 5.4 represent raw values obtained using a random stream, without sending acknowledge packets, and without any data dependencies on the data sent by the other peer. Throughput values are measured for upstream, downstream and aggregate (sum of upstream and downstream). In the remainder of this thesis, the aggregate throughput obtained is used as a reference.

Table 5.4: TL-WR841N 10/100 Mbps Maximum Raw Throughput

	Downstream	Upstream	Aggregate
Throughput	94.54 Mbps	95.88 Mbps	190.42 Mbps

These results show that in practice a throughput close to the nominal network speed has been achieved. One can say the setup introduces a 5% degradation compared to ideal conditions.

The second test consisted in measuring the effective throughput for various packet sizes. Using the `ping` application 10 packets were sent with three distinct data sizes, 54 bytes, 720 bytes and 1440 bytes. The Round-Trip delay Time (RTT) values as reported by the `ping` application and effective throughput for 3 packet sizes are presented in Table 5.5, where the aggregate throughput is calculated using the following expression:

$$Throughput = \frac{(2 * PacketSize * 8)}{RTT * 10^6} \quad (5.1)$$

These results show that the factor limiting the system performance is the latency. The data is packetized and a latency penalty is incurred for each packet sent. Naturally, the penalty decreases with the packet size, but even for the maximum packet size allowed by the CWDP protocol (1440 bytes), the obtained practical upper bound for the aggregate throughput (21 Mbps) is only about 11% of the maximum possible throughput.

Table 5.5: Effective Throughput

Packet Size	Latency	Throughput	Utilization
54 bytes	0.593 ms	1.51 Mbps	<1%
720 bytes	0.84 ms	13.71 Mbps	7.2%
1440 bytes	1.107 ms	20.81 Mbps	10.9%

### 5.3 Throughput Upper Bound Model

Since no alternative paths exist in the network one can say that the One Way Delay (OWD) is approximately  $OWD = \frac{RTT}{2}$ , with  $RTT$  measured by the application. Table 5.6 shows OWD values for the considered packet sizes.

Table 5.6: One Way Delay

Packet Size	Delay
54 bytes	0.297 ms
720 bytes	0.42 ms
1440 bytes	0.554 ms

Using the OWD values, one can calculate lower bound delays for data transactions in the cases of single and double ports. The expression *practical lower bound* is used because the delays measured below were always above these calculations. This model takes into account the handshaking implemented with the small acknowledge packets. Figure 5.1 shows the calculation of the practical lower bound for a single-port FaceWorks system. Sending and receiving back a CORE\_DATA packet takes 1.7 ms.

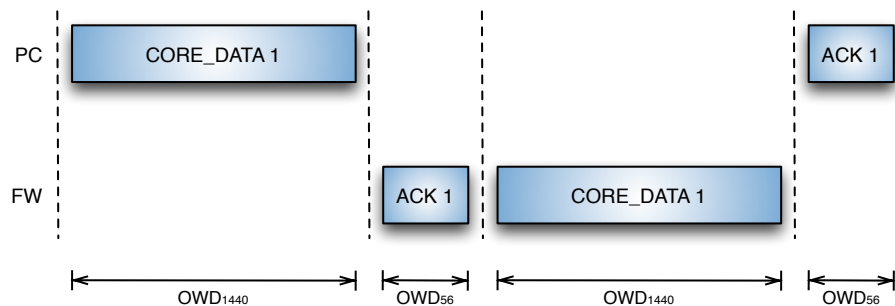


Figure 5.1: Single-Port Timing Diagram

Figure 5.2 shows the calculation of the practical upper bound for a two-port FaceWorks system. Two simultaneously working ports can send and receive back two CORE\_DATA packets in approximately 2.424 ms. The blue color represents the first port, and the red color the second port.

Table 5.7: Throughput Model

	Throughput	Utilization
Single Port	13.55 Mbps	7.11%
Dual Port	19.00 Mbps	10%

Converting lower bound delays to upper bound throughput yields the upper bound throughput results shown in Table 5.7. These results are considerably lower than the switch limits shown

## 5. Results

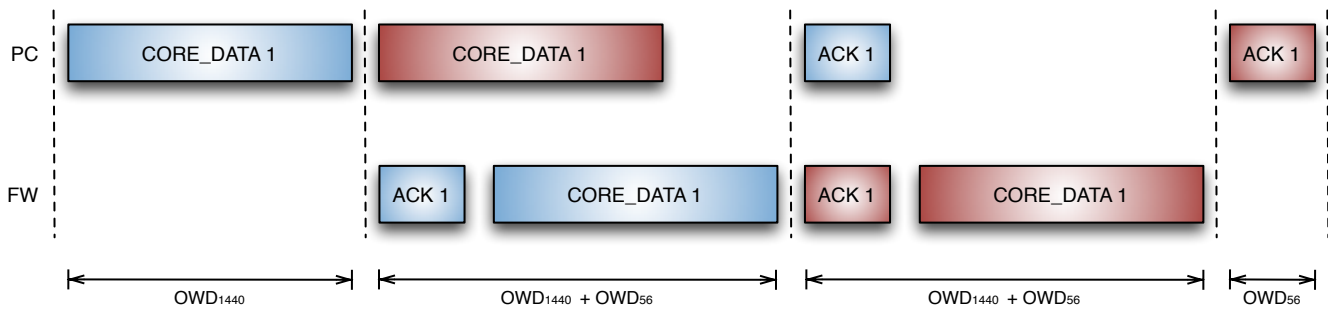


Figure 5.2: Two-Port Timing Diagram

in Table 5.4. This is made clear by column Utilization in the table, which shows the obtained throughput divided by the maximum possible throughput. The explanation for the low utilization rate lies, firstly, in the system latency, and secondly, in the inter packet dependency between the CORE\_DATA packets and the ACK packets.

These results show that the handshaking has a dramatic effect on a single-port FaceWorks, as the next packet can only be sent after the acknowledge for the previous packet is received. The model predicts only about 14 Mbps of effective throughput using a single port.

The situation improves considerably when the second port is added, since a packet for the second port can be sent without waiting for the acknowledge of the packet sent to the first port. The model predicts 19 Mbps of effective throughput, which is close to the 21 Mbps limit obtained without handshaking.

### 5.4 FaceWorks Sustained Throughput

For the sustained<sup>2</sup> throughput measurement, the previously benchmarked TP-Link TL-WR841N 10/100 Mbps switch has been used to interconnect the notebook and the FPGA board containing the FaceWorks core. In the FPGA, the CW-link RX-TX loopback shown in Figure 3.1 was in place, so that the tests could be done using FaceWorks alone. Table 5.8 summarizes the results obtained.

Table 5.8: FaceWorks Throughput Results

	Single Port	Dual Port	Gain
Original FaceWorks	9.86 Mbps	-	-
Current FaceWorks	11.02 Mbps	14.71 Mbps	33%
Verilator Model	42.27 Mbps	79.59 Mbps	88%

Both the original and new FaceWorks implementations have been tested using the Face-Test application. The new version is significantly faster than the original FaceWorks implementation in

<sup>2</sup>In networking the term *sustained* is used for results obtained by averaging over long periods of time.



VHDL. Since the algorithms are the same, the suspicion is that the previous version may have been dropping packets, due to the presence of problems detected by the Verilator linter, which have been fixed in this new and cleaned up Verilog version. Specifically, the original code had a few combinatorial loops which may cause occasional packet losses.

The two-port FaceWorks core has been tested using two Face-Test applications, one for each UDP port. As can be seen, the two-port FaceWorks makes a better use of the available bandwidth when compared to the single-port implementation, as predicted by the throughput model of section 5.3.

Using the cycle accurate Verilator simulator, about 42 Mbps for a single-port core and 80 Mbps for a two-port core have been obtained. Note that, in hardware simulation only, all latencies are zero (network, switch, host and Operating System (OS)), except for the FaceWorks and the MAC protocol small hardware latencies. This clearly demonstrates the dramatic effect of system latency on the board measurements reported before.

The values obtained by the current FaceWorks implementation are compared with the values predicted by the throughput model in Table 5.9. The deviations can be explained by the added latency introduced by the host computer and its operating system. The two-port model performed relatively worse than the single-port model, which can be attributed to the fact that the former requires two Face-Test applications running simultaneously and competing for a shared network interface.

Table 5.9: Measured vs. Modeled Throughput

	Measured	Model	Deviation
Single Port	11.02 Mbps	13.55 Mbps	19%
Dual Port	14.71 Mbps	19.00 Mbps	23%

# 6

## Conclusion

### Contents

6.1 Summary . . . . .	59
6.2 Future work . . . . .	60

## 6.1 Summary

The design of a new multiple port FaceWorks architecture is the main objective of this work, which has been fully accomplished, including its verification and experimental characterization.

FaceWorks is a technology used to test and debug IP cores inside a chip by connecting this chip to an Ethernet network and conducting the tests from a regular Personal Computer (PC). FaceWorks is a patented and proprietary technology of the company Coreworks S.A. FaceWorks uses the UDP/IP protocol stack, topped by the proprietary CWDP layer used for direct interaction with the IP cores being debugged. The IP cores are connected to the FaceWorks module by means of the proprietary CW-Link interfaces.

The IP cores can be tested by test programs running on the PC. The main motivation of this work is to simplify the writing of test software, by allowing multiple threads or processes to exercise multiple IP cores simultaneously using two distinct UDP ports.

The design consisted of extending the FaceWorks hardware to support two ports, and writing a software application to exercise the two ports.

In the hardware implementation, the UDP and CWDP modules have been modified to work with two ports. The number of CW-Link interfaces has been duplicated, in order to keep the same number of debug interfaces per UDP port. Originally FaceWorks was written in the VHDL language. In this work the code has been rewritten in Verilog to allow the new developments to be done also in Verilog. Besides being a lot more common in the industry, the use of Verilog is mandatory to be able to use the Verilator simulator, a key tool which made possible the completion of this project in time.

In the software implementation, a test application (Face-Test) has been developed to send and receive test data to the device under test. This application communicates with the target using sockets, which is useful because it can stimulate both the actual hardware or a simulator.

For verification a cutting-edge approach has been used. Instead of using a standard Verilog testbench and simulator, the Verilator simulator has been used to create a SystemC model of the FaceWorks design, which has been embedded in a C++ testbench. The testbench communicates with the Face-Test application via sockets and drives the FaceWorks SystemC model. This solves the classical problem of integrating software and hardware in a simulation environment. Other advantages of using Verilator are the detailed linting it effects on the Verilog code (due to the fact that it only supports synthesizable code), the exhaustive dumping of wave traces for debugging purposes, and the detailed coverage metrics it extracts from the design simulation.

The design has been synthesized and tested in a Xilinx Spartan 6 FPGA. The implementation proved small and fast enough for a debug core, as such cores should not take too much space in the overall system design. The implementation of the extra port is accomplished at the cost of extra resources. Nonetheless, the resulting FaceWorks IP core is still a small core, using only

## 6. Conclusion

---

10% of the area of an average size FPGA such as the Spartan 6 XC6SLX45T device.

The performance of the core has been measured using the above mentioned FPGA and the Face-Test application. First the characterization of the used Ethernet switch was done to have practical upper bounds for bandwidth, instead of the nominal 100 Mbps. It has been measured that the switch introduces a 5% penalty in the throughput compared to the nominal bandwidth.

Results obtained with the `ping` application show that the factor limiting the system performance is the latency. The data is packetized and a latency penalty is incurred for each packet sent. Naturally the penalty decreases with the packet size but even for the maximum packet size allowed by the CWDP protocol (1440 bytes), the practical upper bound for aggregate throughput thus obtained is of about 21 Mbps.

With the delays obtained with the `ping` application, a model to predict the throughput for Face-Test / FaceWorks transactions has been build. This model takes into account the handshaking implemented with small acknowledge packets. The handshaking has a dramatic effect on a single-port FaceWorks, as the next packet can only be sent after the acknowledge for the previous packet is received. The model predicts only about 14 Mbps of effective aggregate throughput using a single port. The situation improves considerably when the second port is added, since a packet for the second port can be sent without waiting for the acknowledge of the packet sent to the first port. The model predicts 19 Mbps of effective aggregate throughput, which is close to the 21 Mbps obtained without handshaking.

An experimental setup, using a Face-Test application sending and receiving packets to a single-port FaceWorks core equipped with a CW-Link loopback, showed that, in practice, only about 11 Mbps aggregate throughput can be obtained, compared to the 14 Mbps as predicted by the model. Another experiment, using two Face-Test applications sending and receiving packets to a two-port FaceWorks core, achieved about 15 Mbps, compared to the 19 Mbps predicted by the model. This difference can be explained by the added latency introduced by the host computer and its operating system. In an extreme situation where all latencies were zero (network, switch, host and OS), one could obtain about 42 Mbps for a single-port core and 80 Mbps for a two-port core; this result has been obtained using just the cycle accurate Verilator model.

## 6.2 Future work

To improve the FaceWorks throughput one obvious solution is upgrading it to use a 1 Gbps PHY chip. This would allow for a lower latency because of a higher transmission rate and a higher MTU with the use of Jumbo frames. However, implementing support for Jumbo frames would require larger amounts of on-chip RAMs, and this could be a restriction depending on the target FPGA device.

### 6.2.1 Not Acknowledge

This solution consists in not sending acknowledge packets; if some packet is lost or received out-of-order then the retransmission of that packet is requested. This solution is based on the assumption that most packets sent from the PC to FaceWorks and vice-versa will be delivered and arrive in order.

Such mode would work in the following way. The controlling host (PC) starts by registering with the FaceWorks using the command `SET_CORE_ACCESS`. This command is replied with an ACK packet. This is necessary so the PC knows that at least the first packet has been received and that it is now controlling FaceWorks. After this, ACK packets are disabled for `CORE_DATA` packets. Instead, when the receiver detects a packet whose sequence number is not  $n + 1$ , where  $n$  is the sequence number of the previous packet, it emits a Not acknowledge (NACK) requesting the retransmission of the lost packet  $n+1$ .

Figure 6.1(a) shows an example transfer from the PC to FaceWorks, where packet 3 is lost. After receiving packet 4 and dropping it, FaceWorks sends a (NACK packet for packet 3 to the PC. All subsequent packets are dropped until packet 3 is received. When packet 3 is received, the transmission resumes.

Figure 6.1(b) shows an example transfer from the PC to FaceWorks where packet 4 is delivered before packet 3. The same recovery procedure is performed as described previously.

However, the NACK solution requires a more complex software application to control the packet transmission rate, since the receipt of too many NACK packets may signal the presence of network congestion. In this case the rate of transmission should be lowered to avoid congestion. Reversely, the transmission rate can be augmented if the incidence of NACK packets is not increased.

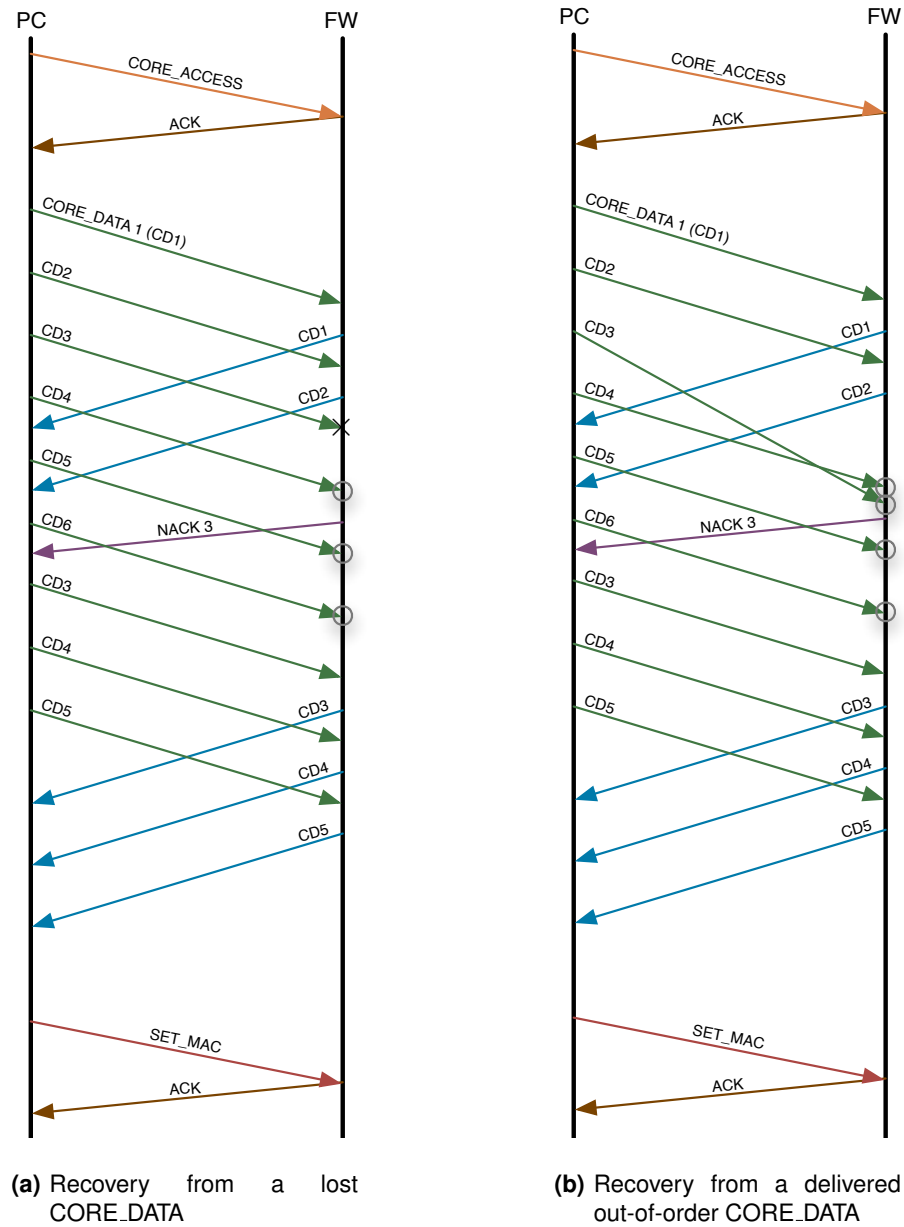


Figure 6.1: Packet Loss Example Recoveries using NACK's

### 6.2.2 A UDP/CWDP Cycle Accurate Simulator

The solution proposed previously for the FaceWorks simulator was designed to debug FaceWorks core itself. To debug an IP core having a CW-Link interface, one could simply *verilate* it together with FaceWorks. However, if the objective is to debug the IP core only, simulating FaceWorks may be an unnecessary burden. A simpler alternative is presented in Figure 6.2. It consists in replacing the full FaceWorks model with a simple UDP/CWDP server in the C/C++ testbench.

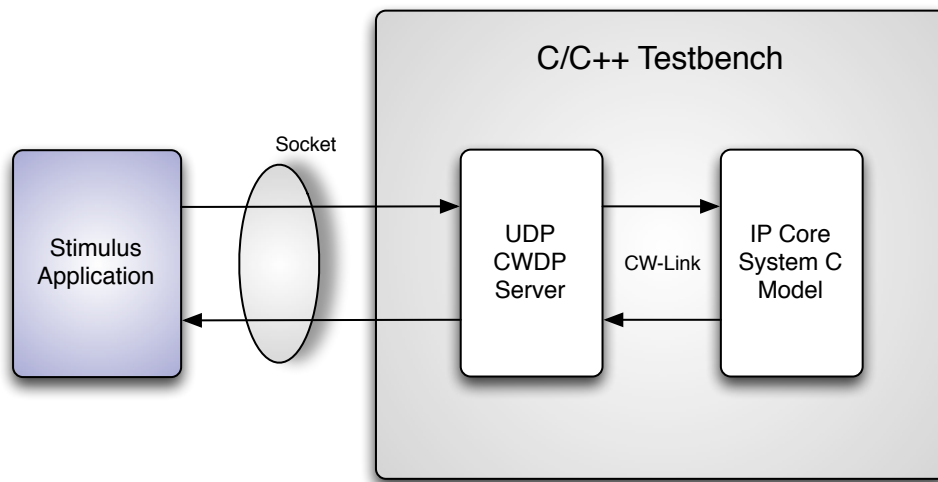


Figure 6.2: A UDP/CWDP Cycle Accurate Simulator

The UDP/CWDP server interacts with the IP core model using CW-Link interfaces. Test applications like Face-Test can be written in any language as long as they use network sockets for communication and implement the CWDP protocol. These test application can later be reused to test the same IP core in a chip that contains the FaceWorks core. Finally, the major advantage of this approach is the fact that the simulation of the IP core becomes simpler and faster because the FaceWorks model is not being simulated.

## 6. Conclusion

---



# Bibliography

- [1] J. Hussein, M. Klein, and M. Hart, "Lowering power at 28 nm with xilinx 7 series fpgas," [http://www.xilinx.com/support/documentation/white\\_papers/wp389\\_Lowering\\_Power\\_at\\_28nm.pdf](http://www.xilinx.com/support/documentation/white_papers/wp389_Lowering_Power_at_28nm.pdf), Xilinx Inc, Tech. Rep., February 2012.
- [2] Wikipedia, "Geforce 600 series," [http://en.wikipedia.org/wiki/GeForce\\_600\\_Series](http://en.wikipedia.org/wiki/GeForce_600_Series), June 2013.
- [3] —, "Transistor count," [http://en.wikipedia.org/wiki/Transistor\\_count](http://en.wikipedia.org/wiki/Transistor_count), June 2013.
- [4] A. Shimpi and R. Smith, "The intel ivy bridge core i7-3770k review," <http://www.anandtech.com/show/5771/the-intel-ivy-bridge-core-i7-3770k-review/3>, April 2012.
- [5] A. L. Shimpi, "The amd trinity a10-5800k a8-5600k review," <http://www.anandtech.com/show/6332/amd-trinity-a10-5800k-a8-5600k-review-part-1>, September 2012.
- [6] J. Gelas, "The xeon e5-2600 dual sandybridge for servers," <http://www.anandtech.com/show/5553/the-xeon-e52600-dual-sandybridge-for-servers/2>, March 2012.
- [7] A. Shimpi, "The bulldozer review amd fx8150 tested," <http://www.anandtech.com/show/4955/the-bulldozer-review-amd-fx8150-tested>, October 2011.
- [8] J.-M. Yannou, "Xilinx's 3d (or 2.5d) packaging enables the world's highest capacity fpga device and one of the most powerful processors on the market," [http://www.i-micronews.com/upload/pdf/3DPackaging\\_Nov.2011\\_AC.pdf](http://www.i-micronews.com/upload/pdf/3DPackaging_Nov.2011_AC.pdf), November 2011.
- [9] J. de Sousa, N. Lourenço, N. Ribeiro, V. Martins, and R. Martins, "Network core access architecture," Patent US 2008/0 288 652, 05 15, 2008. [Online]. Available: <http://www.google.com/patents/US8019832>
- [10] —, "Network core access architecture," Patent EP 2 003 571/A2, 12 17, 2008. [Online]. Available: <http://www.google.com/patents/EP2003571A2>

## Bibliography

---

- [11] I. S. Association, IEEE Standard for Ethernet. IEEE, August 2012, ch. Section 2, Chapter 22 Reconciliation Sublayer (RS) and Media Independent Interface (MII).
- [12] —, IEEE Standard for Ethernet. IEEE, August 2012, ch. Section 1, Chapter 3 Media Access Control (MAC) frame and packet specifications.
- [13] S. Bradner, “Key words for use in rfcs to indicate requirement levels,” <http://www.ietf.org/rfc/rfc2119.txt>, March 1997.
- [14] D. C. Plummer, “An ethernet address resolution protocol,” <http://www.ietf.org/rfc/rfc768.txt>, November 1982.
- [15] J. Postel, “Internet protocol,” <http://www.ietf.org/rfc/rfc791.txt>, September 1981.
- [16] —, “User datagram protocol,” <http://www.ietf.org/rfc/rfc826.txt>, August 1980.
- [17] Coreworks, FaceWorks CWnet01 Datasheet, Multi-purpose Autonomous Network Interface Core. Coreworks, January 2008.
- [18] W. Snyder, “Verilog simulator benchmarks,” [http://www.veripool.org/wiki/veripool/Verilog\\_Simulator\\_Benchmarks](http://www.veripool.org/wiki/veripool/Verilog_Simulator_Benchmarks), March 2011.
- [19] —, “Verilator 3.846 manual,” <http://www.veripool.org/projects/verilator/wiki/Documentation>, March 2013.
- [20] T. Carstens, “Sniffex, sniffer example of tcp/ip packet capture using libpcap,” <http://www.tcpdump.org/sniffex.c>, June 2013.
- [21] T. Pircher, “Cyclic redundancy check (crc) calculator and c source code generator,” [http://www.tty1.net/pycrc/index\\_en.html](http://www.tty1.net/pycrc/index_en.html), April 2013.
- [22] T. Bybell, “Gtkwave, waveform viewer,” <http://gtkwave.sourceforge.net>, April 2013.



## **Appendix A GTKwave**



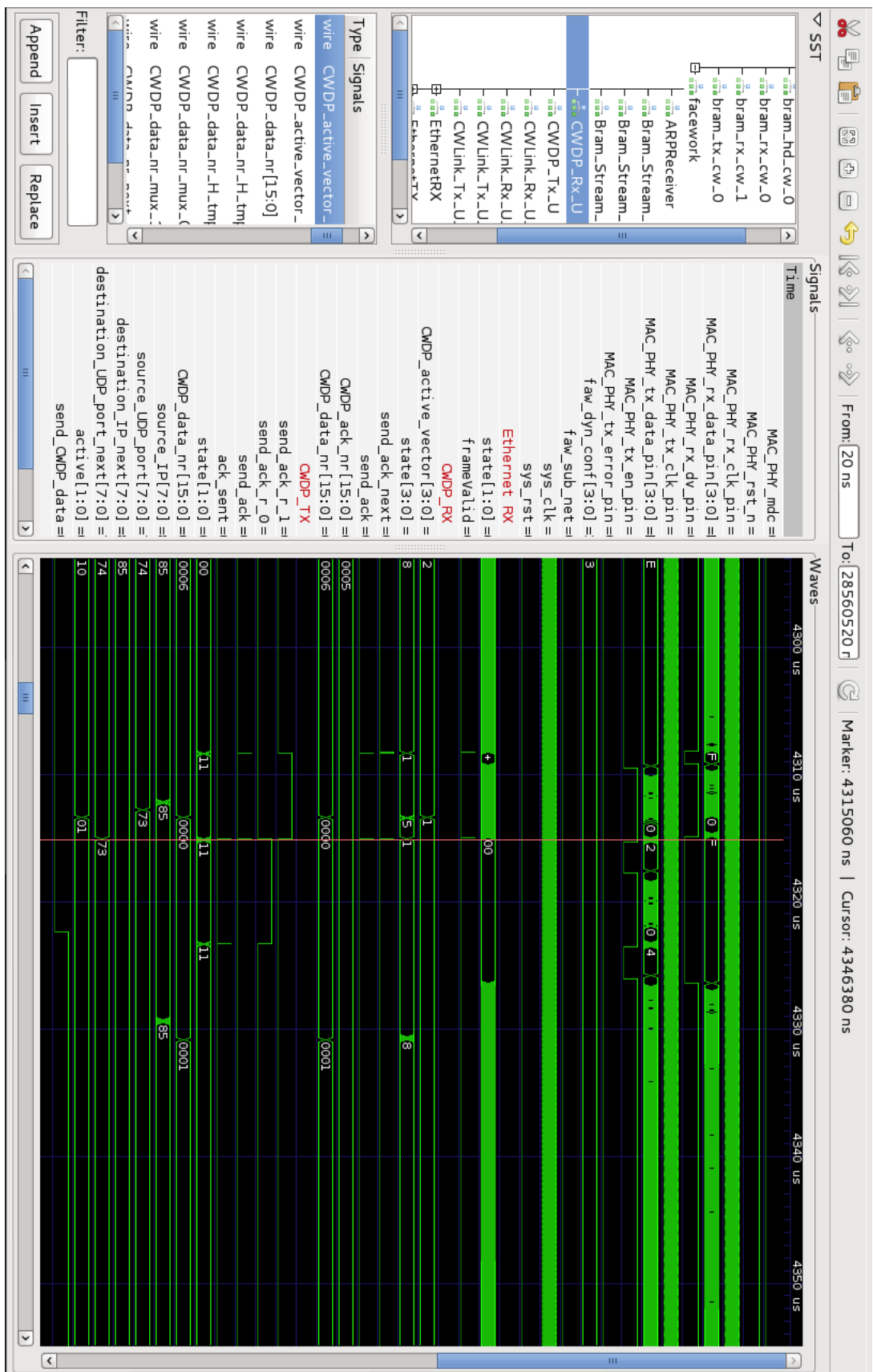


Figure A.1: A GTKwave Screenshot Showing FaceWorks Signals