
The University of Queensland

# Implementation of a High Performance PCI Express UART using an FPGA

Student Jordan CONLEY

| | |
|---|---|
| Course Code | ENGG7290 |
| Submission Date | 28 June 2018 |
| Academic Supervisor | Prof. Neil Bergmann |
| On-site Supervisor | Toby Smith |
| Placement Institution | Opengear |

Faculty of Engineering, Architecture and Information Technology

# Executive Summary

RS-232 serial ports can be found on many industrial devices, network appliances, and servers. Serial console servers contain dozens of RS-232 ports for managing and monitoring this hardware. This provides an *out-of-band* management solution which is not otherwise affected by network issues or outages and can be used as a last-resort troubleshooting tool when such hardware or infrastructure fails.

RS-232 is a physical layer specification for connecting Universal Asynchronous Receiver Transmitter (UART) ports to one another which includes handshaking and flow-control. In many of Opengear's current products, RS-232 ports are driven by Exar XR17V358 PCI Express UART chips. Currently, there are few suppliers of PCI Express UART chips and this poses a credible business risk for Opengear. Additionally, the current solution using Exar PCI Express UARTs is both expensive and slow.

This project seeks to replace the current solution with a custom *field programmable gate array* (FPGA) based solution with the aim of improving system performance and providing a credible alternative for Opengear in the event that Exar were to cease production of their UART chips.

The FPGA UART is divided into three main parts: the FPGA design, a printed circuit board (PCB) to expose the physical RS-232 signals, and a Linux driver to allow user applications to operate the serial ports. The driver accepts data from user applications through Linux character devices, packs it into buffers compatible with the FPGA, and maps these buffers to the system bus memory. The FPGA contains a DMA core which reads these buffers and sorts the contained data into a series of FIFOs, one for each UART port.

For data flow in the receive direction, data read on each of the FIFOs is placed into a shallow FIFO to be eventually packed into a much larger shared buffer. The use of a single, shared, buffer makes the design more efficient in cases where some ports are used more than others – there are no FIFOs left empty while others are potentially overflowing. It also simplifies the process of preparing data for the DMA core. When data arrives in the shared buffer, or when the buffer fills beyond a threshold, the FPGA sends an interrupt request to the host CPU, causing the driver to empty this buffer and deliver its contents to the relevant applications.

Benchmarks were taken by automating the use of Opengear's internal tool `serbench`.

`serbench` takes a list of serial ports and writes data through them, measuring the total throughput and other statistics. The automation script runs `serbench` for every number of ports up to 96. Statistics about CPU utilisation and interrupt request counts are taken between `serbench` samples. This provides a snapshot of how the UART system performs.

Benchmarking the FPGA UART directly against Exar UARTs is infeasible. The Exar UARTs cannot be connected to the development computer without expensive intervening hardware, and the Opengear console server being used as a reference point does not have an available PCI Express port. Other Opengear console servers which use PCI Express connectors internally are not compatible with standard PCI Express.

The FPGA UART was instead benchmarked using the development computer. In an effort to make the comparison more balanced, the development computer, which contains a quad-core Intel CPU running at 2.8 GHz was down-clocked to the minimum supported speed of 1.6 GHz, configured to use a single core only. It was also placed under an artificially high load.

It was found that even after placing the CPU under an exceptionally high load, and running it at a reduced speed, the FPGA UART still greatly outperformed the original CM7196's Exar UARTs. The increase to performance is partly owed to the new design's approach to interrupt triggering, which leads it to fire two orders of magnitude less interrupt requests than the CM7196.

The goal of this project was primarily to provide a proof of concept for an FPGA-based high-performance UART. By re-factoring parts of this design, it would be fairly straightforward to adapt this work for use in a production environment. The most obvious change to make would be to shift the architecture from having a single monolithic FPGA with shift registers for I/O pins to using several smaller FPGAs. By using smaller FPGAs with less UART ports each, the design can be easily scaled down to be useful on Opengear's smaller console servers as well as the CM7196.

# Acknowledgements

I would like to take this opportunity to acknowledge the team at Opengear and the academic staff at UQ for their support in bringing this project to life. In particular,

- Neil Bergmann
- Toby Smith
- Ken Wilson
- Ashkan Boldaji
- Tim Gibson
- Jacqui Porteus
- Tony Merenda
- David Leonard

# Contents

# List of Figures

# List of Tables

# Section 1

# Introduction

## 1.1 Motivation

Opengear designs, manufactures, and sells serial console servers and network management appliances. These can have between 4 and 96 RS-232 serial ports to provide *out-of-band* monitoring services for network equipment and servers. Central to these servers is an array of PCI Express connected Universal Asynchronous Receiver/Transmitters (UARTs).

Currently, there is only one major manufacturer of PCI Express UART controllers, Exar Corporation, and this poses a significant risk to Opengear. If Exar were to cease production of its UARTs, Opengear would be left with very few options for replacement.

Additionally, there are severe performance issues with the Exar parts currently in use. Notably, the Exar parts use memory-mapped transfers over PCI Express to communicate with the console server CPU. This leads to extreme overhead with large volumes of data, and causes performance issues. On top of this, the Exar UARTs create a storm of CPU interrupts, further degrading performance. This is explored in Section 2.1.

Aside from being slow, the Exar solution is also expensive. For a 96 port unit, 6 PCI Express lanes are needed. This means external PCI Express switches are required to connect all of the UART controllers. The actual Exar parts are also expensive individually.

For all of these reasons, it is proposed that the current solution be replaced with a *field programmable gate array* (FPGA) based solution. An FPGA would connect via PCI Express to the host CPU and control all of the RS-232 ports. The FPGA solution uses Direct Memory Access (DMA) to transfer data efficiently, rather than relying on slow direct memory mapped transfers.

.

## 1.2 Project Goals and Scope

To be successful, the FPGA design must

1. implement a PCI Express multi-port UART, for up to 96 ports, and
2. demonstrate improved performance over the previous solution.

Although not key requirements for the project, ideal stretch goals are for it to be

1. cheaper than the current solution,
2. able to incorporate extra peripherals such as ethernet, and
3. able to be optimised in order to fit on a smaller device than the proposed development platform.

## 1.3 Deliverables

To achieve these goals, each of the following deliverables have been developed:

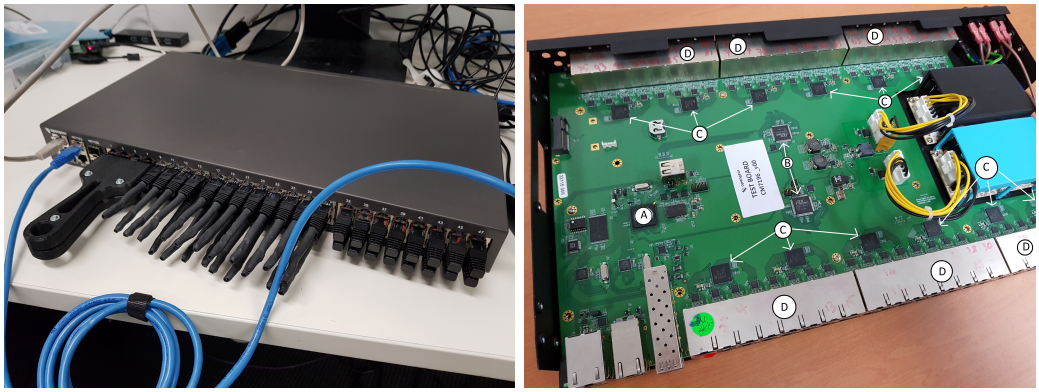
1. An FPGA design implemented using VHSIC Hardware Description Language (VHDL) and Intellectual Property cores (IP).
2. A driver for Linux to communicate with the FPGA.
3. A HDL testbench to verify the correctness of the design.
4. A suite of tools and scripts used to benchmark the design.
5. Complete design documentation for each of the aforementioned deliverables, in the form of this report.

# Section 2

# Technical Background and Literature Review

## 2.1 Opengear CM7196 Console Server

The Opengear CM7196 Console Server is shown in Figure 2.1. It uses 96 RJ45 connectors, 48 each on the front and back, as RS-232 serial ports. In the figure, serial loop-back plugs are connected to each of the 96 ports for benchmarking. This server allows users to connect on a web interface or using SSH to remotely access servers and networking equipment *out of band*. In this context, *out of band* refers to the use of secondary networks including cellular and 4G.




(a) External view (b) Internal view

Figure 2.1: Images of the Opengear CM7196 96-port Console Server.

Figure 2.1b shows the inside of the device. The CM7196 comprises of an ARM CPU (callout A) connected via PCI Express switches (B) to 12 eight-port PCI Express UARTs (C). Each of these octal UARTs controls a second octal UART, for a total of sixteen ports per PCI Express lane. The UARTs connect to the RJ45 ports (D) via RS232 transceivers. This is detailed in Figure 2.2.

The PCI Express UART used in the CM7196 is the Exar XR17V358. In order to

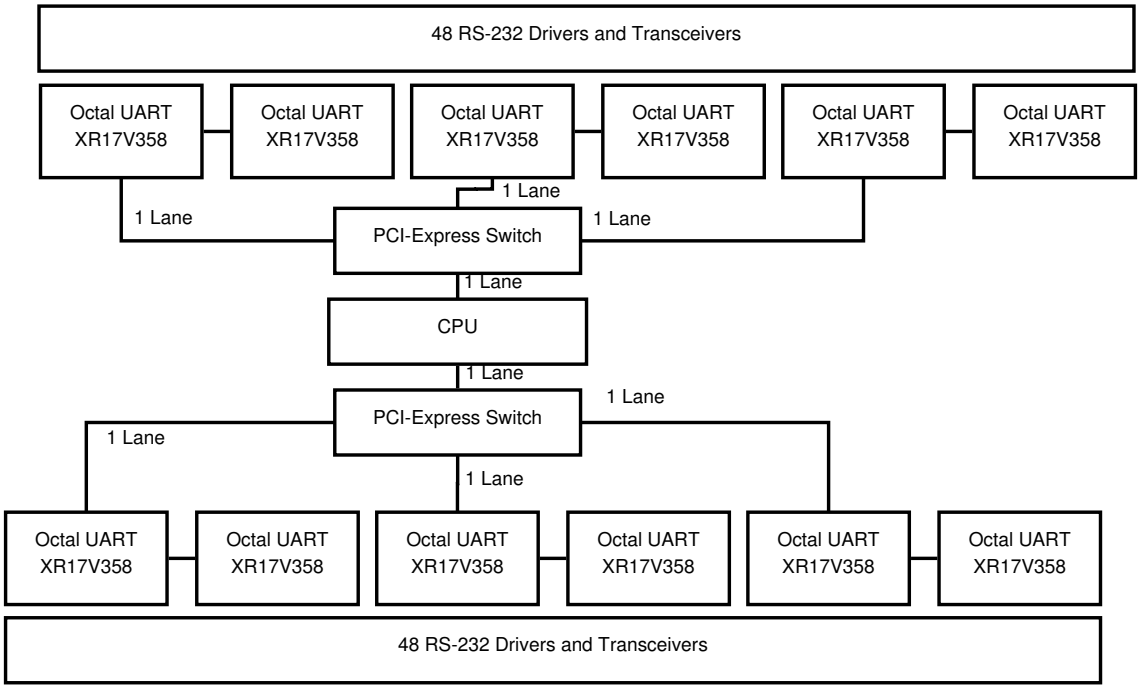


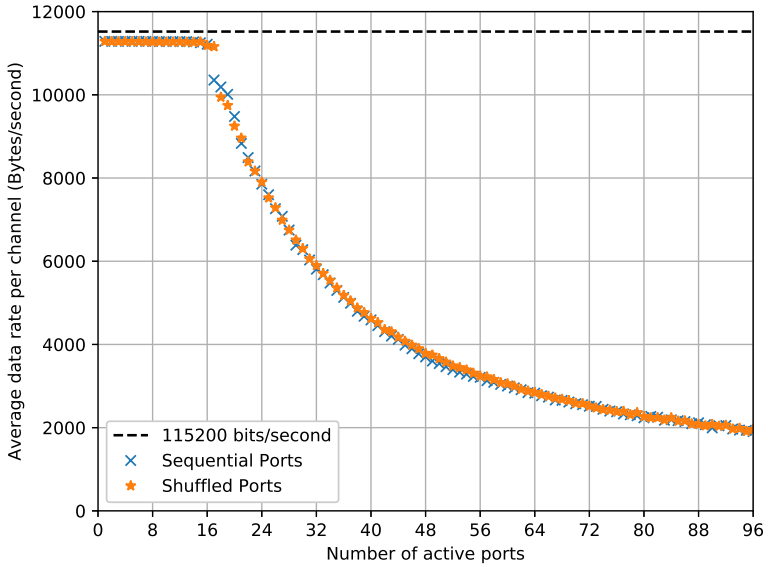

Figure 2.2: Diagram showing the current configuration of PCI Express UARTs in the Opengear CM7196 serial console.

maintain compatibility with existing hardware drivers, the XR17V358 implements a 16550-compatible interface (see Section 2.4). Data is transmitted and received through 256 byte first-in-first-out buffers (FIFOs), which are filled and emptied using memory-mapped reads and writes over PCI Express [1].

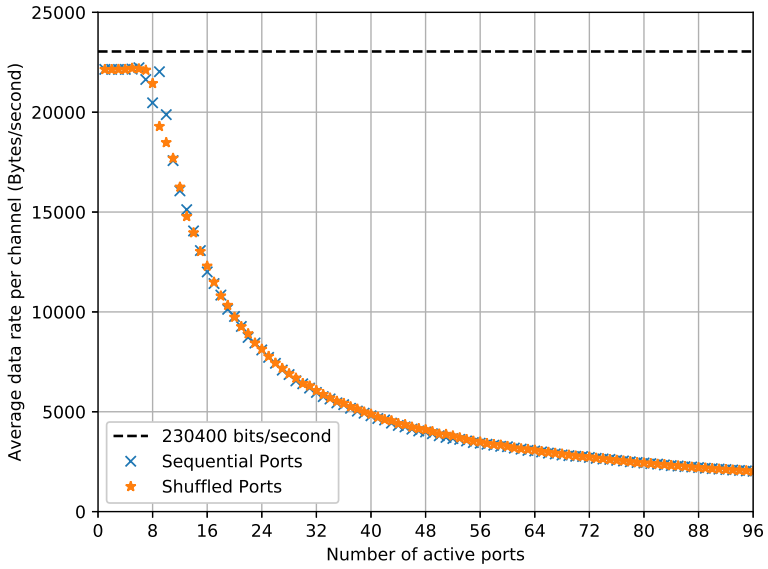
Unfortunately, the current system is unable to maintain the maximum data rate in configurations where more than a handful of ports are being operated simultaneously. The CM7196 console server when operating under *benchmark conditions* (see Section 3.5) is unable to completely saturate more than 16 ports at 115 200 baud or 8 at 230 400 baud.

Figure 2.3 shows that as the number of active serial ports increases, average throughput through each port decreases. In tests including both sequential and shuffled ports, there is a severe drop-off in performance when there are more than a few active ports. There is a negligible difference in performance between the shuffled and sequential port tests. Figure 2.4 provides a possible explanation for this: above 16 or 8 ports respectively, the CPU is at almost 100% utilisation.

As the number of active ports increases, Figure 2.3 shows the system tends towards a maximum total data rate: around 192 kB/s at 115 200 baud and 200 kB/s at 230 400 baud. When only a few ports are active and each port is able to be operated at the maximum baudrate, that becomes the limiting value. The slight increase in total data rate between 115 200 baud and 230 400 baud is likely because in the time between interrupt requests being handled at 230 400 baud, there is more data in the UART receive FIFO and more space in the UART transmission FIFO, and as such the driver can process more data per




(a) 115 200 baud

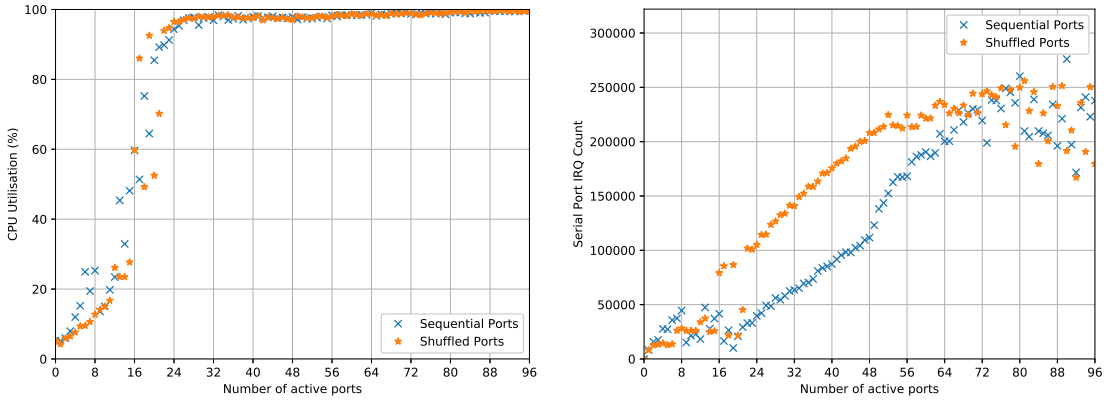



(b) 230 400 baud

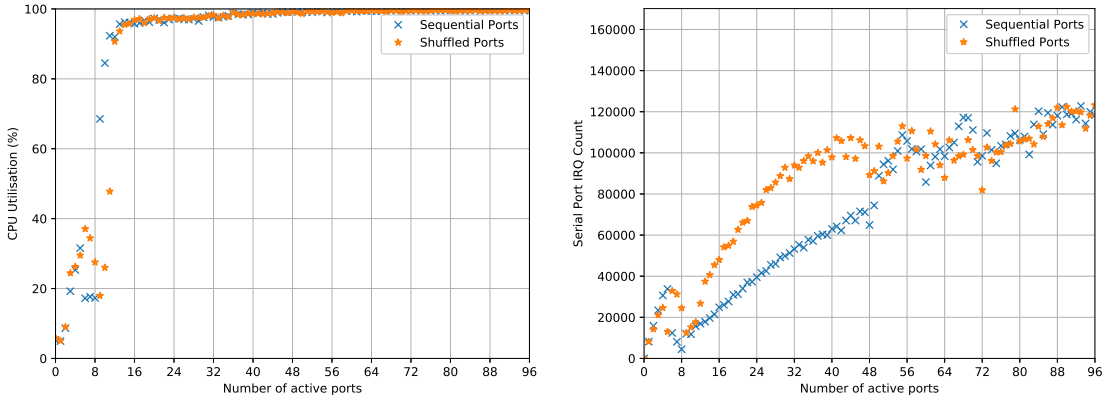
Figure 2.3: Average data rate per channel for a CM7196 Serial Console.

interrupt.

Figure 2.4 shows the number of serial port interrupts which are received by the CM7196's CPU during the performance tests. When sending 1 MB each through 96 serial ports at 230 400 baud, about 120 000 interrupts are received. This corresponds to about 840 B of total throughput per interrupt, or about 9 B per port. This is most likely responsible for the slow data rates observed. About twice as many interrupts are generated at 115 200 baud, which explains the decrease in total throughput.




(a) 115 200 baud




(b) 230 400 baud

Figure 2.4: CPU utilisation for $N$ active channels for an CM7196 Serial Console.

## 2.2 Universal Asynchronous Receiver / Transmitters

The Universal Asynchronous Receiver/Transmitter (UART) signal protocol in its basic form is quite simple. Two asynchronous simplex connections connect two devices to perform a serial transfer of bytes. For communication between UARTs to work, each pair of receiver and transmitter must be preconfigured to operate with the same bit rate, parity settings, and byte width. UART has no concept of high-level communication, only of the transfer of bytes.

A UART *frame* is shown in Figure 2.5. The idle state for a UART with the line driven high, and the start of a transmission is signalled by driving it low – the start-bit. Bits are then transmitted, from least significant bit to most significant bit. An optional parity bit can be appended. A logic-high stop-bit signals the end of the transmission. There is no requirement for a break between transmissions, and another transaction can appear immediately after the stop-bit.

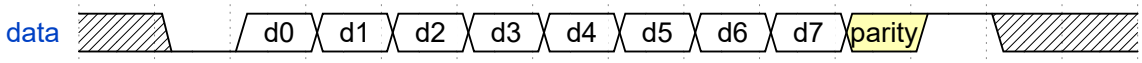
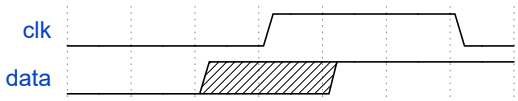


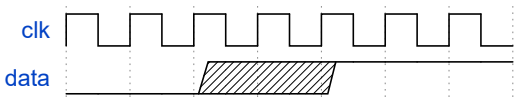

Figure 2.5: A single 8-bit transmission using UART signalling.

The bit clock is not typically transmitted in a normal UART, instead both the transmitter and receiver typically independently produce their own baud clocks [2]. UART signals are often transmitted using voltage signalling standards such as RS-232, RS-422, or RS-485 for communication between devices, and these standards are responsible for defining flow-control and handshaking.[1] [3], [4]

The absence of a reference bit clock creates challenges for the UART receiver. The receiver must create its own clock, and compensate for any variation in frequency between the transmitter's clock and its own. Additionally, if the clocks are only slightly phase misaligned, the receiver may sample an output which is metastable (Figure 2.6a). To solve this problem, typical UART implementations oversample the data line 8 or 16 times the baudrate and use some form of bit averaging to decode the input, as in Figure 2.6b. [2], [3] This also allows a UART to recover from a small frequency offset in the bit clock.




(a) Single clock per bit: Potential for sampling a metastable input




(b) Multiple clocks per bit: Can recover from an unstable input.

Figure 2.6: Comparison of UART receiver schemes.

A single UART may contain many ports, where a port contains a pair of receive and transmit lines. For instance, the Exar XR17V358 is referred to as an 8-port UART.

[1] Despite this, UART implementations will often include RTS/CTS flow control, even if they don't include the rest of the RS-232 signals. The Microchip MCP2200 USB/UART bridge is an example of this.

## 2.3 Oxford Semiconductor OXPCIe954

Oxford Semiconductor, before its acquisition by PLX Technologies, produced a series of PCI Express serial ports. In particular, the *OXPCIe954 PCI Express Bridge to Quad Serial Port* [5] until its obsolescence in 2013 [6].

Similar to the Exar XR17V358, the OXPCIe954 is capable of controlling two *secondary* parts. As a result, it is possible to connect up to 12 RS-232 ports to a single PCI Express lane [5].

Unlike the Exar XR17V358 and 16550-compatible FPGA IP cores, the OXPCIe954 has the ability to use Direct Memory Access (DMA) over PCI Express to populate and empty its FIFOs, as rather than memory-mapped transfers. The result is a more efficient use of the available PCI Express bandwidth, and presumably lower CPU overhead. In order to take advantage of the DMA features, Oxford Semiconductor provides drivers for Windows and Linux. [5]

Oxford Semiconductor also produced the OXPCIe958 *PCI Express Bridge to Octal Serial Port*. Similarly to the XR17V358, the OXPCIe958 can control a single twin part for a total of 16 UARTs per PCI Express lane. [7]

## 2.4 National Semiconductor 16550 UART

Many commercially available UARTs are at least somewhat compatible with the 16550, originally produced by National Semiconductor. The 16550 itself was a successor to the 16450 and 8250 UARTs. The principle advantage over the 8250 was the inclusion of receiver and transmitter FIFOs. These FIFOs meant that the CPU was not required to service interrupts generated by the UART immediately, thereby improving system performance. The 16550 is now produced by Texas Instruments, since their acquisition of National Semiconductor, as the PC16550D. [3]

The 16550 sets out the minimum feature set for the project. Specifically,

- 5, 6, 7, and 8 bit bytes,
- at least 16 byte FIFOs,
- parity bit generation in even, odd, and stick modes,
- variable baudrate control,
- flow control using RTS/CTS and DTR/DSR pairs, and
- maskable interrupts for
  - data being received,
  - the transmission FIFO being empty, and
  - a change in any of the incoming flow-control and handshake signals.

Additionally, the PC16550D datasheet describes how each of the handshaking and flow

control signals interact with each other.[2]

- The idle state for each of the flow control signals is logic high (pulled-up).
- When another device is connected to a port, the Data Carrier Detect (DCD) pin is connected directly to ground – logic low. This alerts the UART that a physical connection to another device has been made.
- When either host is ready to establish a connection, they assert their Data Terminal Ready (DTR) pin low, and await a logic low in response on the Data Set Ready (DSR) pin.
- Once two hosts have established a connection using DTR/DSR and one wishes to send data, the Request to Send (RTS) pin is asserted low. A logic low response is then awaited on the Clear to Send (CTS) pin.

## 2.5 PCI Express

PCI Express is a high-performance packet-switched point-to-point network, much like ethernet. A *network* of PCI Express devices emulate a bus, which is largely compatible with the original Peripheral Component Interconnect (PCI) bus it replaces. The main components of a PCI Express bus are *links*, *switches*, and *endpoints* which combine as in Figure 2.2. In this particular design there is a CPU acting as a *root* and PCI Express switches which create extra lanes to connect more *end-points*. Under Linux, information about the PCI Express bus and connected devices can be retrieved using the `lspci` utility or read from `sysfs` [8].

The PCI Express stack is made up of three layers:

- the physical layer,
- the link layer,
- the transaction layer.

On the physical layer, PCI Express is comprised of differential, serial point-to-point simplex links. An opposing pair of simplex links are paired to form lanes. Each lane is capable of transmission at either 2.5 GT/s or 5 GT/s. When this bandwidth is insufficient, extra lanes can be combined up to a total of 32 PCI Express lanes per link. Total bandwidth scales linearly with the number of lanes. [9], [10]

The link layer reassembles packets received by the physical layer and performs error checking to ensure they were received correctly. In the case where multiple PCI Express lanes are used, the link layer removes any skew between the incoming lanes. Finally, it maintains a buffer to resend any packets which were flagged as corrupted upon arrival. This guarantees reliable transmission of data. [9], [10]

[2]This datasheet is not the original standard by which these signals are defined, however it is more important for the design to be compatible the 16550 than the official RS-232 standard.

The transaction layer converts link layer packets into a format more readily accessible by the host device. It is also responsible for creating link layer packets based on software requests. Software requests are typically created by drivers on the host. [9]

## 2.6 Field Programmable Gate Arrays

Field Programmable Gate Arrays (FPGAs) are a form of re-programmable integrated circuit (IC) which can be used to implement digital circuits. They offer maximum flexibility in digital designs without losing the performance and customisation which is normally associated with fully custom ICs[11], [12] like ASICs[3]. The name *field-programmable gate array* stems from the fact that an FPGA can be seen as an array of reprogrammable logic gates[11].

Internally, FPGAs are realised using an array of logical blocks connected by network of interconnections [12]. On Xilinx FPGAs, these are known as *configurable logic blocks* (CLBs) and include lookup tables, memory elements (flip-flops), arithmetic logic, and multiplexers [13]. The arrangement of logic blocks and interconnections is often referred to as *fabric*. To ease development and improve performance, FPGAs also include pre-built components for common functionality ranging from block RAM, multi-gigabit tranceivers, and PCI Express *hard blocks*. [10], [14]

The main advantage in using an FPGA for this project over a processor or microcontroller is the performance. By using an FPGA, it is easy to implement an array of UART state machines which are completely independent of one another. The nature of a digital circuit implies that they are entirely parallel.

For common logic functions which do not need to be implemented in hard logic, FPGA vendors such as Xilinx and Altera implement and license *intellectual property* (IP) cores. As distinct from hard blocks (sometimes called *hard IP*), *soft* IP cores are implemented in the fabric of an FPGA. Examples of soft IP cores are explored in sections 2.7 and 2.8.

FPGAs are ubiquitously programmed using vendor-specific software such as Xilinx *Vivado*[15], Intel *Quartus Prime*[16], or Lattice *Diamond*[17] using hardware description languages such as VHDL or Verilog in conjunction with IP cores.

## 2.7 Altera/Intel IP

Altera/Intel[4] intellectual property cores (IP) are primarily connected using the *Avalon* family of interfaces. Most importantly, the Avalon Streaming Interface (Avalon-ST) and Avalon Memory Mapped Interface (Avalon-MM). As the names suggest, Avalon-ST is

[3] Application-specific integrated circuits

[4] Intel Corporation acquired Altera Corporation at the end of 2015 [18]. Documentation for Altera FGPAs and IP cores has largely been re-branded by Intel to reflect this, and as such for all intents and purposes, they can be considered one and the same.

designed to transfer continuous streams of data, whilst Avalon-MM is designed for high performance memory mapped communications. [19]

On devices which contain dedicated PCI Express logic, the Cyclone, Arria, and Stratix devices, Intel makes the IP to use it available at no cost. For the purposes of this project, we consider the Cyclone V FPGA platform. The *Cyclone V Hard IP for PCI Express* can be configured to provide either an Avalon-ST or an Avalon-MM interface to the FPGA design. [10]

Intel also provides two separate IP cores for the UART interface: the *Intel FPGA 16550 Compatible UART Core* and the *UART Core*. Both of these IP cores use an Avalon-MM slave interface to communicate with other peripherals. The *UART Core* is unsuitable for this project since it does not support the DCD, DTR, and DSR signals. Both of these cores require a paid license to use.

## 2.8 Xilinx IP

FPGA manufacturer Xilinx provides intellectual property (IP) cores to cover both UART and PCI Express functionality. The majority of IP in Xilinx' catalogue communicates using the *Advanced Microcontroller Bus Architecture Advanced eXtensible Interface 4* (AXI4) protocol[20].

### 2.8.1 AXI Interfaces

The AXI4 specification defines three standards for connecting modules together: AXI4, AXI4-Lite, and AXI4-Stream. AXI4-Lite is a subset of the full AXI4 connection, requiring only a minimal number of signals to be connected, and is suitable for low-throughput communications. AXI4 is designed for high performance memory mapped transfers, and AXI4-Stream is for continuous streams of data. [20]–[22]

AXI4 and AXI4-Lite interfaces are point-to-point links which form buses using *interconnects*. Xilinx provides interconnect IP which is capable of routing any number of masters to any number of slaves using AXI3, AXI4, and AXI4-Lite, automatically handling issues regarding clock domain crossing. [20]

#### AXI4-Lite

AXI4-Lite uses 5 independent channels for data transmission, each containing a READY/-VALID pair for handshaking. The list of compulsory channels and signals is shown in Table 2.1. If an AXI master wishes to read from an slave address, it sends that address on the *read address* channel, and awaits a response on the *read data* channel. Similarly, the master can write an address and then data in the *write address* and *data* channels, and then await a response on *write response*. [22]

To transfer data on a single channel, the driving source (AXI master or slave) sets the payload (ARADDR, RDATA, AWADDR, etc.) and asserts the corresponding VALID signal. The driver must then wait for the receiver to assert the READY signal, if it was not already asserted. Data is latched by the receiver on the rising edge of the clock when both VALID and READY are high. An example of this *handshake* is shown in Figure 2.7. Specifically, this is a valid-before-ready handshake. It is possible for both ready and valid to be asserted together, or for ready to be asserted before valid, so long as they only remain asserted together for no more than one clock cycle[5]. [20], [22]

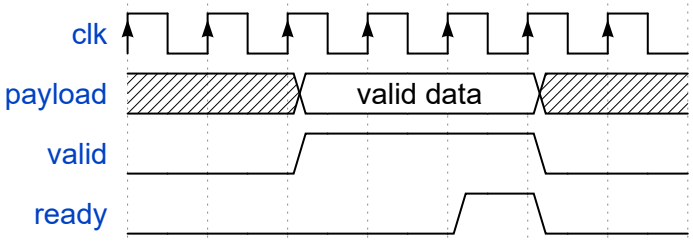



Figure 2.7: An example of a correct AXI handshake. [22]

| Read Address | Read Data | Write Address | Write Data | Write Response |
|---|---|---|---|---|
| ARADDR[31:0] | RDATA[31:0] | AWADDR[31:0] | WDATA[31:0] | BRESP[1:0] |
| ARREADY | RDREADY | AWREADY | WREADY | BREADY |
| ARVALID | RVALID | AWVALID | WVALID | BVALID |
| – | RRESP[1:0] | – | WSTRB[3:0] | – |

Table 2.1: Signals required for the AXI4-Lite interface. This project does not make use of the extra transaction security and user signals, so they have been omitted.

### AXI4-Stream

AXI4-Stream is a similar interface which is useful when the address mapping features of AXI4 or AXI4-Lite are not necessary. An AXI4-Stream connection is similar to the *write data* channel from AXI4-Lite. Data is transferred unidirectionally using the same READY/VALID handshake, however extra signals are also available. These signals are summarised in Table 2.2. Unlike AXI4-Lite, AXI4-Stream is able to perform multiple transfers side-by-side following the same rule as before – on a rising edge of the clock when both READY and VALID are asserted, a transfer occurs.

Unlike AXI4-Lite, the width of most AXI4-Stream fields are not directly specified by the standard. For instance the data bus may be any whole number of bytes (N in Table 2.2) wide, and TKEEP and TSTRB each contain 1 bit for each byte in TDATA. The TKEEP

[5]This is a requirement of AXI4-Lite, but not of AXI4. In AXI4, sequential transfers are permitted and an extra signal marks the end of a transmission

signal allows data channels to support partially *full* transfers, which is useful when creating an interface between two streams of different widths.

| Mnemonic | Description | Driver |
|---|---|---|
| TDATA[8N-1:0] | Data bus | Master |
| TUSER[U-1:0] | User side-band data bus | Master |
| TDEST[D-1:0] | Data routing information | Master |
| TKEEP[N-1:0] | Keep flag to identify unused bytes | Master |
| TSTRB[N-1:0] | Strobe flag to distinguish position bytes from data | Master |
| TLAST | Signifies the end of a transmission | Master |
| TID[I-1:0] | Identifies data streams | Master |
| TVALID | Valid handshake | Master |
| TREADY | Ready handshake | Slave |

Table 2.2: List of AXI4-Stream signals. [21]

The AXI4-Stream only supports simplex communication on its own [21] so some IP cores, including the Xilinx DMA IP core use pairs to create duplex links [23].

### 2.8.2 PCI Express IP

In the Xilinx 7-series FPGAs, the Artix, Kintex, and Virtex families all contain dedicated hardware for handling PCI Express connectivity. For devices which include this hardware, Xilinx provides three IP cores for integrating in a larger design. These are

- 7 Series Gen2 Integrated Block for PCI Express [14],
- AXI Memory Mapped to PCI Express [24], and
- DMA/Bridge Subsystem for PCI Express [23].

All three of these IP cores are licensed with the Xilinx Vivado Design Suite.

The *Integrated Block for PCI Express* is targeted at lower-level applications than the AXI Memory Mapped and DMA IP cores. It provides a conversion layer between PCI Express transactions and AXI-Stream. An extra layer of logic is required to create a memory mapped or DMA interface. Both the AXI Memory Mapped and DMA IP cores contain the Integrated Block for PCI Express.

The *AXI Memory Mapped to PCI Express* IP provides a conversion layer between memory mapped reads and writes between PCI Express and AXI [24]. Concretely, the IP core forwards read and write requests on any of the PCI Express memory spaces to the AXI bus. This would be a suitable choice if the project aim was purely limited to memory mapped transfers, however is unsuitable given DMA is required.

The *DMA/Bridge Subsystem for PCI Express* implements both an I/O memory interface and a scatter-gather DMA interface for PCI Express. Using the scatter-gather DMA engine, the IP core can be configured to read from a series of discontiguous buffers in host memory,

saving the overhead of having to copy multiple buffers together. For this reason, the Xilinx DMA (XDMA) IP core has been used in the project. [23]

### 2.8.3 16550 UART IP

Xilinx also provides an IP core which implements a 16550-compatible UART [25]. The *AXI UART 16550* processes reads and writes to an identical interface as the original 16550 UART using the AXI4 bus. This includes memory mapped reads and writes to the on-board RX and TX FIFOs. As discussed above, using memory mapped reads and writes to FIFOs, as opposed to DMA, is inefficient. This memory mapped interface alone makes the IP core unsuitable for use in this project.

The product guide for the IP core provides an estimate of the FPGA resources required to implement the core. For comparison, a very simple UART with full support for flow control, automatic FIFO reads and writes, and near feature-parity with the 16650 was been implemented. Table 2.3 shows a comparison between the FPGA resources required to implement the Xilinx 16650 IP core, the simple UART, and the final UART developed in the project.

| Resource | Xilinx IP Core | | Simple UART | Final UART |
|---|---|---|---|---|
| | 16550 Mode | 16450 Mode | | |
| Slices / CLB | 120 | 82 | 52 | 57 |
| Registers | 318 | 259 | 111 | 114 |
| Look-up tables | 352 | 252 | 197 | 129 |

Table 2.3: Comparison of FPGA resource utilisation between the Xilinx 16550 IP [25] and a preliminary implementation.

The discrepancy between the Xilinx implementation and the preliminary one stems from the fact that the 16550 IP is designed to be entirely standalone. It generates its own baud clock from the AXI interface clock, and it implements its own complete AXI4 interface. The design incorporating the preliminary UART, and the overall design for this project achieves high efficiency by sharing many of these components. For instance, rather than having 96 ports each with their own AXI4 interface, there are 96 ports with one shared AXI4 interface.

## 2.9 CMOS Power Modelling

The final PCB for this project when fully populated contains almost 100 74-series logic ICs, each running at clock speeds of 18.432 MHz or 36.864 MHz. As part of the design process, a power budget was constructed in order to verify that the PCB power supply is capable of providing the required power. To accurately predict the device's power draw, it is necessary to understand *how* the devices use power.

Devices fabricated using complementary MOSFET (CMOS) technology draw negligible current during static operation, and only require meaningful amounts of current while switching. When a logic circuit implemented with CMOS technology changes state, there is a brief moment during the transition where both the N-channel and P-channel FETs are conducting, effectively creating a short circuit between the 2.5 V rail and ground. There is also a brief current draw to *charge* the parasitic gate capacitance in the MOSFETs. [26]–[28]

These two sources for dynamic current draw are modelled using the *power dissipation capacitance*, $C_{PD}$ [26]. $C_{PD}$ provides a frequency-dependent model for the dynamic power consumption of the device, and is an empirically measured value, typically on the order of 50 pF, provided in the datasheets for CMOS ICs. Thus, the dynamic power dissipation contributed by $C_{PD}$ for a CMOS device with supply voltage $V_{DD}$ and switching frequency $f$ is $V_{DD}^2 C_{PD} f$.

To complete the model of dynamic power draw from a CMOS IC, the power used driving the outputs must be modelled. By modelling each of an IC's outputs as having a capacitance of $C_L$ being switched at $f_{out}$, the complete dynamic power dissipation of a CMOS device becomes

$$P_D = V_{DD}^2 C_{PD} f + \sum \left( C_L V_{DD}^2 f_{out} \right)$$

## 2.10 Linux Kernel Driver Development

A module for the Linux kernel will be implemented to allow user applications to operate the UART hardware. This module communicates directly with the Xilinx DMA IP configuration memory and provides a standard interface for user applications.

### 2.10.1 PCI Express and DMA Subsystem

The Linux kernel provides an API to simplify developing drivers for devices which communicate via PCI Express. When a module is loaded into the kernel, it registers a list of device it can potentially control. The module then registers a set of PCI Express device callback operations using `pci_register_driver`. When the kernel detects a PCI Express device that the module may be able to control, it uses the `probe` method registered by `pci_register_driver` to *offer* the device to the module. Drivers are expected to perform all of the necessary device setup in the `probe` method. [8]

Copying large blocks of data from user applications to PCI Express devices using the I/O memory interface is remarkably inefficient in terms of CPU time (see Section 2.1), and as such DMA is often used instead. DMA allows the target device to read data directly from memory without the CPU having to waste cycles manually transferring data. [8], [29]

To transfer memory blocks using DMA, they must be in an address space which is accessible by the PCI Express bus. Fortunately, the Linux kernel provides APIs for

allocating memory which is DMA-addressable, and for mapping already allocated memory to DMA addresses. [8]

To use the XDMA IP core, the driver must create a list of *descriptors* which contain the bus address and size for a series of transfers. The memory layout for the XDMA descriptors is shown in Table 2.4. Descriptors create a linked list structure using the address field. This list is then itself transferred to bus memory, and the address of the first descriptor is passed to the XDMA IP core by writing to I/O memory. Creating multiple transfers and having the hardware assemble them in this way is called scatter-gather DMA. [8], [23], [29]

| Offset | Field |
|---|---|
| 0x00 | [0:7] Control byte<br>[8:13] Adjacent descriptor count<br>[14:15] Reserved<br>[16:31] Magic value for checksum |
| 0x04 | Transfer length |
| 0x08 | Source address (low word) |
| 0x0C | Source address (high word) |
| 0x10 | Destination address (low word) |
| 0x14 | Destination address (high word) |
| 0x18 | Next descriptor address (low word) |
| 0x1C | Next descriptor address (high word) |

Table 2.4: XDMA descriptor format [23]

### 2.10.2 Character Devices

Character (`char`) devices provide a *file*-like interface to the driver for user applications. At a base level, reading, writing, and memory mapped I/O operations are able to be implemented by a kernel module using a `char` device interface. This lends itself very nicely to being used to represent serial ports. [8]

Once loaded, `char` devices are identified using two numbers: the major and minor device numbers. A single major device number corresponds to a single driver, and minor device numbers can create different endpoints for that driver. A list of registered `char` devices and their major number in the kernel can be found in `/proc/devices`. Once the major number for a device is known, a node in the filesystem can be created using the `mknod` system call, or command of the same name. Filesystem nodes can also be created from within the driver. [8]

# Section 3

# Project Methodology and Management

Careful time management and planning meant that this project was able to be completed on-time, despite several time-consuming setbacks. Weekly meetings with on-site supervisors to discuss project progress helped ensure that milestones and expectations were consistently being met.

## 3.1 Project Activities

The key activities and initial rough plan for this project were as follows.

1. Determine the necessary scope and resources for the project, and acquire relevant hardware and development tools.
2. Establish a benchmarking procedure (see Section 3.5)
3. Benchmark and analyse existing Opengear hardware (CM7196, IM7248).
4. Design and implement the three key components:
   (a) FPGA configuration including testbench,
   (b) 96-port PCB including DC power budget, and
   (c) Linux driver, implemented first in userspace, then as a kernel driver.
5. Benchmark the final solution, and investigate performance improvements as required.

As the project progressed, the list of activities expanded to include an 8-port PCB, and exclude the userspace Linux driver.

## 3.2 Final Project Timeline

A Gantt chart showing the final project timeline is presented overleaf.

As outlined in Section 4.3, the scope for the project expanded to include the design and manufacture of a second PCB. This forced the 96-port PCB to be put on hold.

Additionally, unforeseen delays with the PCB manufacturer meant that the 96-port board did not arrive until very late[1] in the project. Fortunately, the design had already been validated due to the 8-port board, and benchmarking could be carried out using an internal hardware loop-back in the FPGA design, so this did not cause other work to be delayed.

It was originally expected that the Linux driver could first be implemented as a userspace application and then adapted to be a full kernel driver. This proved to be impossible because the Xilinx driver relied on the DMA IP core receiving extraneous side-band signals from the hardware which were not implemented. It was then decided to reallocate the time which was to be used for the user-space driver to be used instead for implementing a complete kernel driver. Xilinx has since released a new driver for the DMA IP.

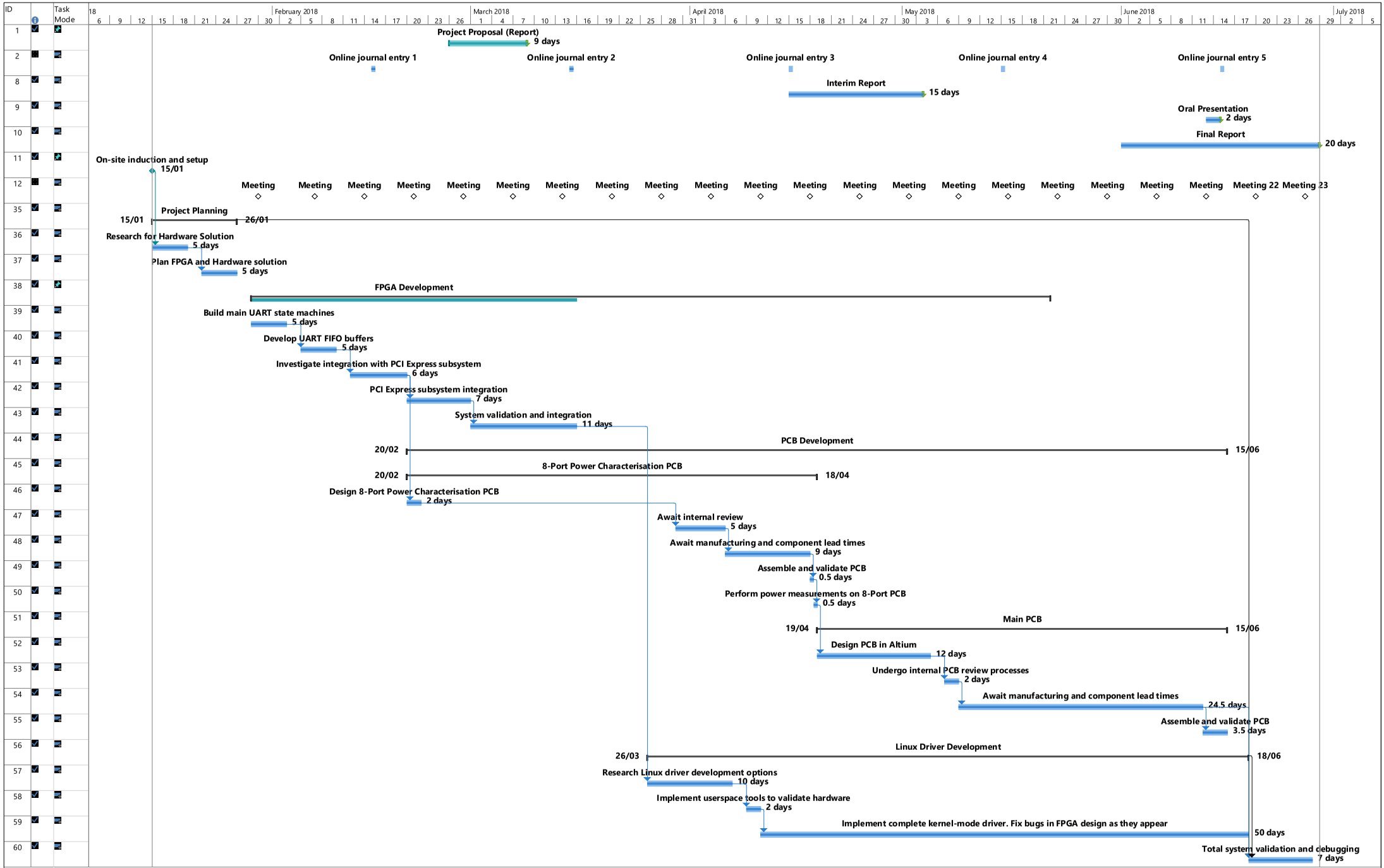
## 3.3 Project Milestones

Project progress is monitored by ensuring certain key milestones are met. These are outlined in Table 3.1. Progress towards each of these milestones was assessed in the weekly on-site supervisor meetings. Naturally, all of these milestones have now been met.

| Description | Date | Status |
|---|---|---|
| Start | 15/1/18 | Project start date. Research for the project commences on this date. |
| Initial progress meeting | 29/1/18 | Completed on time. |
| Project proposal | 8/3/18 | Completed on time. |
| Completion of FPGA and hardware design | 15/3/18 | Completed, minor tweaks were made to improve performance and fix bugs until early June. |
| Interim report | 3/5/18 | Completed on time. |
| Completion of driver development | 8/6/18 | Completed, tweaks were made to improve performance and fix bugs until late June. |
| On-site presentation | 27/6/18 | Completed on time. |
| Final thesis due | 28/6/18 | Completed on time. |

Table 3.1: List of key project milestones.

[1] Week 22


ID
Task Mode
February 2018
March 2018
April 2018
May 2018
June 2018
July 2018
Project Proposal (Report)
9 days
Online journal entry 1
Online journal entry 2
Online journal entry 3
Online journal entry 4
Online journal entry 5
Interim Report
15 days
Oral Presentation
2 days
Final Report
20 days
On-site induction and setup
15/01
Meeting
Meeting 22
Meeting 23
Project Planning
15/01
26/01
Research for Hardware Solution
5 days
Plan FPGA and Hardware solution
5 days
FPGA Development
Build main UART state machines
5 days
Develop UART FIFO buffers
5 days
Investigate integration with PCI Express subsystem
6 days
PCI Express subsystem integration
7 days
System validation and integration
11 days
PCB Development
20/02
15/06
8-Port Power Characterisation PCB
20/02
18/04
Design 8-Port Power Characterisation PCB
2 days
Await internal review
5 days
Await manufacturing and component lead times
9 days
Assemble and validate PCB
0.5 days
Perform power measurements on 8-Port PCB
0.5 days
Main PCB
19/04
15/06
Design PCB in Altium
12 days
Undergo internal PCB review processes
2 days
Await manufacturing and component lead times
24.5 days
Assemble and validate PCB
3.5 days
Linux Driver Development
26/03
18/06
Research Linux driver development options
10 days
Implement userspace tools to validate hardware
2 days
Implement complete kernel-mode driver. Fix bugs in FPGA design as they appear
50 days
Total system validation and debugging
7 days

## 3.4 Project Resources

In order to complete the project, the following resources were required.

- A Xilinx AC701 Artix-7 FPGA development kit
- Xilinx Vivado, the accompanying integrated development environment (IDE) required for programming the FPGA.
- Altium Designer, to design the PCBs.
- The capability to manufacture, assemble, and debug a large and complex printed circuit board.
- A development environment to develop software and drivers for Linux. This includes a test machine with a spare PCI Express slot and riser cable with at least 4 lanes.
- Opengear CM7196 and IM7248 console servers, to use as reference points for benchmarking.

## 3.5 Benchmarking Methodology

Benchmarking was carried out by automating the use of Opengear's internal tool `serbench`. Originally created to test key performance requirements, the purpose of `serbench` is to push as much data as possible through a set of serial ports in loop-back configuration and measure

- time taken to send data,
- loop-back latency,
- average data rate per port, and
- approximate bit error rate.

This provides a snapshot of the UART's performance.

Benchmarks were taken by running `serbench` on $1 - N$ ports, up to the number of ports on the device. From that each of the above metrics were obtained. Tests were conducted using both sequential and shuffled serial ports. When using shuffled ports, the testing script selects ports such that the number of Exar UARTs being exercised is maximised for a given number of ports.

Between measurements, kernel and system statistics are logged from `/proc/stat` and `/proc/interrupts`. Specifically time spent

- in user mode,
- in system mode,
- waiting for I/O to complete, and
- idle

are measured. From these measurements, it is possible to determine approximately how much CPU time is spent operating the UARTs, and how well the system performs.

A script automates the running of `serbench` and extraction of kernel statistics. It was run externally to the console server, from a Linux computer with an Intel quad-core CPU. Controlling the test externally meant that processing the data from `/proc/stat` and `/proc/interrupts` could be done without impacting the system being tested. It also meant that the networking layer on the device need not be used, reducing unnecessary CPU utilisation.

The *loopback latency* statistic taken from `serbench` is disregarded entirely. The figure is meaningless because of internal buffering effects in the drivers. To solve the issue, the driver would have to artificially throttle incoming data, creating an unnecessarily slow input barrier. It was also necessary to remove this *feature* to reduce memory consumption when run on the CM7196.

Bit error rate was also ignored, because using RTS/CTS flow control meant that errors were almost entirely neutralised. On the CM7196, the only time bit errors were introduced after the introduction of flow control was when a loop-back plug was not inserted properly.

## 3.6 Risks and Opportunities Analysis

There were few risks in this project, and the vast majority of the risks which were identified were able to be mitigated almost entirely. Additionally, Opengear stands to benefit greatly from the project's outcomes.

### 3.6.1 Risks for Completion

All of the following risks, if they were to eventuate, would have stood to become major issues for on-time completion.

Most notably, it is possible that after all the development work with the FPGA, it is simply not possible to extend performance far enough due to factors not relating directly to the implementation itself. The main source of this risk is the CPU being used.

The project plan calls for expensive equipment to develop with. It is possible that some of this equipment may become accidentally damaged during the course of the project. Damaged equipment would almost certainly require replacement. This is still an ongoing risk which is minimised by moving expensive equipment out of reach and inside enclosures while not in use.

As FPGA designs become larger and more interconnected, they require faster and higher capacity FPGAs to host them. Naturally, there was the risk that the chosen FPGA would have been too small to use. In this project, the Artix-7 XC7A200T FPGA was chosen as the host, because it is much larger than it was expected to be necessary.

The 96 port PCB requires 6 layers for routing. Issues with the design or manufacturing, could cause the project would require a new board to be built. This could seriously prevent the project from being completed on time, and potentially be very expensive. A smaller, 8

port PCB was developed to validate the design to help catch any potential issues early.

Issues with manufacturing and shipping the 96 port PCB could have meant that by the time it arrived, it was too late to be useful. The 8-port PCB pushing back the manufacturing for this board increases the risk. This was partially mitigated by requesting express shipping and rapid turnaround on manufacture. Unfortunately, this did eventuate, and the board did not arrive until quite late in the project. The effects of this were minimal, because the FPGA design included an internal loop-back function, which emulated the presence of the board for benchmarking.

### 3.6.2 Health and Safety Related Risks

There are inherent risks associated with any electronics work. Fortunately, this particular project is safe assuming appropriate care is taken. The three main sources of personal risk are as follows.

1. The risk of injuries arising from incorrectly using a soldering iron when building the PCB. This has been mitigated using common sense and after having undergone training in safely operating this equipment.
2. The risk of receiving an electric shock from exposed electronics within the testing setup. This has been eliminated entirely by using a PCI Express riser to bring the testing set-up outside the host computer's case, so there is no need to reach inside near any kind of dangerous voltages.
3. The risk of static shock arising from the use of otherwise harmless electronics. This is exacerbated by dry weather, as is common during the winter months. The mostly harmless shocks, whilst annoying, were eliminated through the use of an electro-static discharge (ESD) wristband, grounded with the testing equipment.

### 3.6.3 Commercial Issues

Although this project is not expected to be moved into production without significant revision, there would still be commercial risks associated with the current design if it were to be used in production.

Most significantly, there is a risk that the hardware configuration for the FPGA is compromised. Many FPGAs require external memory to store the configuration *bitstream*. This risk can be mitigated through the use of encrypted bitstreams.

Developing hardware which requires ball-grid array (BGA) or other difficult to solder packages are an inherent manufacturing risk, because it is difficult to ensure that all pins are correctly connected without an X-ray or significant testing. FPGAs are ubiquitously sold on BGA packages and so moving the design into production would require soldering these packages. This is considered a negligible risk, given that Opengear currently manufactures hardware using such packages. For the purposes of prototyping, and as a risk-reduction strategy, all BGA packages were eliminated using extra breakout boards.

The final design utilises a large number of ICs for inputs and outputs. Every added IC is an extra component which can fail or be added incorrectly and is therefore a risk. Given that 74-series logic ICs are relatively simple and come in easy to solder packages, this is a fairly small risk. The design would have to be changed significantly to alleviate the issue.

### 3.6.4 Commercial Opportunities

This project provides several opportunities for Opengear.

Perhaps the most significant opportunity for Opengear is that now there is a credible alternative to the Exar XR17V358 for PCI Express UARTs. Additionally, by combining many serial ports into a single FPGA and a few commodity logic chips, it is possible to greatly reduce the overall cost to manufacture.

FPGAs are, by design, reprogrammable. By using an FPGA for the project, it is therefore possible to patch any issues with the design in the form of a firmware update. Implementing this would require additional hardware to reprogram the FPGA bitstream memory, however this hardware may already need to be included anyway for manufacturing.

The main deliverable for this project is a complete PCI Express high-performance multi-port UART. The PCI Express component of the project is handled by an Xilinx IP core, and the actual multi-port UART component interfaces with that using AXI interfaces. As such, the new sections of tehe design could be used in a other systems, such as with Xilinx Zynq CPU+FPGA system-on-chip which provides AXI interfaces for peripherals. This would allow Opengear to reduce the number of components in smaller devices which don't need dozens of UARTs.

# Section 4

# System Design

A high performance 96-port PCI Express UART was constructed using an FPGA. This UART was benchmarked against the CM7196's UART (see Section 2.1), and a significant improvement in performance has been seen.

## 4.1 Design Overview

The overall design is as follows. A custom printed circuit board (PCB) containing 96 RS-232 ports connects to an FPGA development kit. The FPGA then interfaces with a host PC running Linux via PCI Express. Finally, a Linux kernel driver allows user applications to use the UART ports using character devices for serial interfaces. Figure 4.1 shows this configuration.

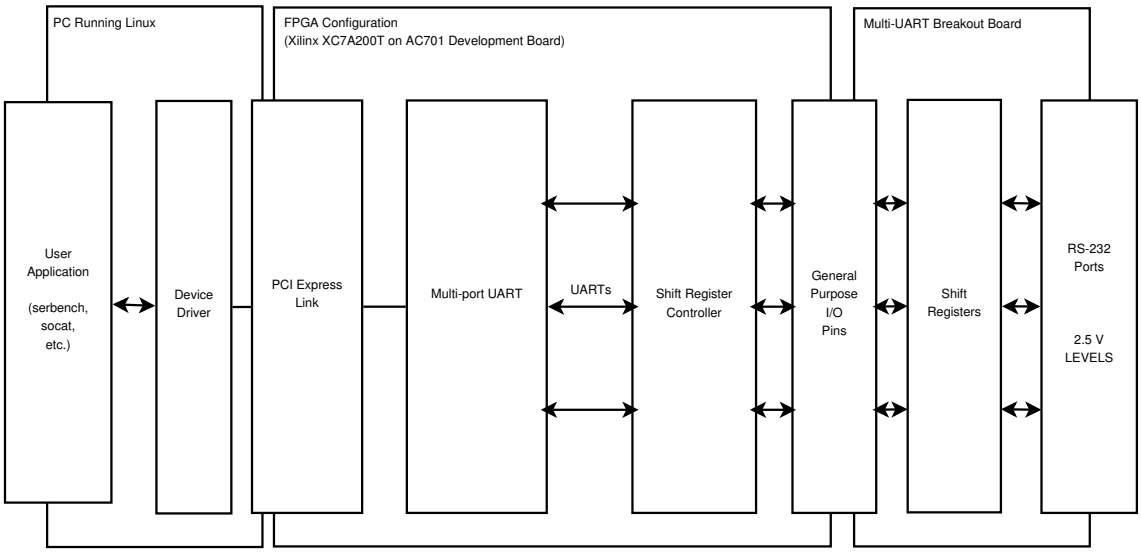



Figure 4.1: System diagram for the project.

The *RS-232 Ports* label is perhaps a misnomer, because the signals are output at 2.5 V levels, rather than the actual RS-232 voltage levels used in production. By not including

RS-232 line drivers, this prototype is made considerably cheaper, and able to be tested using a simple logic analyser. Since the purpose of the line drivers is to set the correct voltage levels, their omission does not have an impact on the correctness of the signalling.

Perhaps the only meaningful effect resulting from omitting the line drivers is that now the *input* signals (RXD, CTS, DSR, and DCD) need to have their idle state pulled high using resistors.

## 4.2 FPGA

### 4.2.1 Overview

A Xilinx Artix-7 FPGA development kit, the AC701 has been used to host the design. It was chosen because it has a high density FMC connector for general-purpose input and output (GPIO) and a 4-lane PCI Express edge connector, without being overly expensive. [30]

Figure 4.2 provides an overview for the FPGA component of the design. The design interfaces with the PCI Express interface using a Xilinx IP core (*DMA/Bridge Subsystem to PCI Express*). This IP core exposes two AXI interfaces: AXI-Lite and AXI4-Stream[1].

The configuration registers, FIFOs, signal controller array, and shift register array are all synchronous to a clock running at 73.728 MHz – the UART clock. The DMA-AXI interface operates at 125 MHz, requiring the use of interconnect IP. The device is required to support baudrates up to 230 400 baud, and it is desirable to be able to oversample the inputs and outputs at a factor of at least 8 times – a sampling rate of 1.843 MHz. In order to achieve this sampling speed through shift registers, and to accommodate the internal FIFO logic, a clock rate of 40 times the sampling speed is used – 73.728 MHz.

The AXI4-Lite interface is used for memory-mapped reads and writes to the configuration memory component in the design. This area provides a bank of registers to allow the driver to control and monitor the UART signal controller and FIFOs. The current memory layout for registers here can be found in Appendix A. Currently, options are included for baudrate selection, and flow control. Interrupt masking is handled by the Xilinx IP core.

Data is received by the design from the DMA IP Core through AXI4-Streams. Data to be sent out, or received by, the serial ports is transmitted in a 32-bit format (See Table 4.1). This format allows each individual character to be treated as a separately routable packet.

| Bit Range | Description |
|---|---|
| 31 to 24 | Destination Port |
| 23 to 8 | Reserved for status (break, parity, XON/XOFF flow control, 9th data bit, etc.) |
| 7 to 0 | Serial data. |

Table 4.1: Bit allocation for FIFO data packets

[1]The AXI4-Stream interface appears as two interfaces on the diagram for convenience.

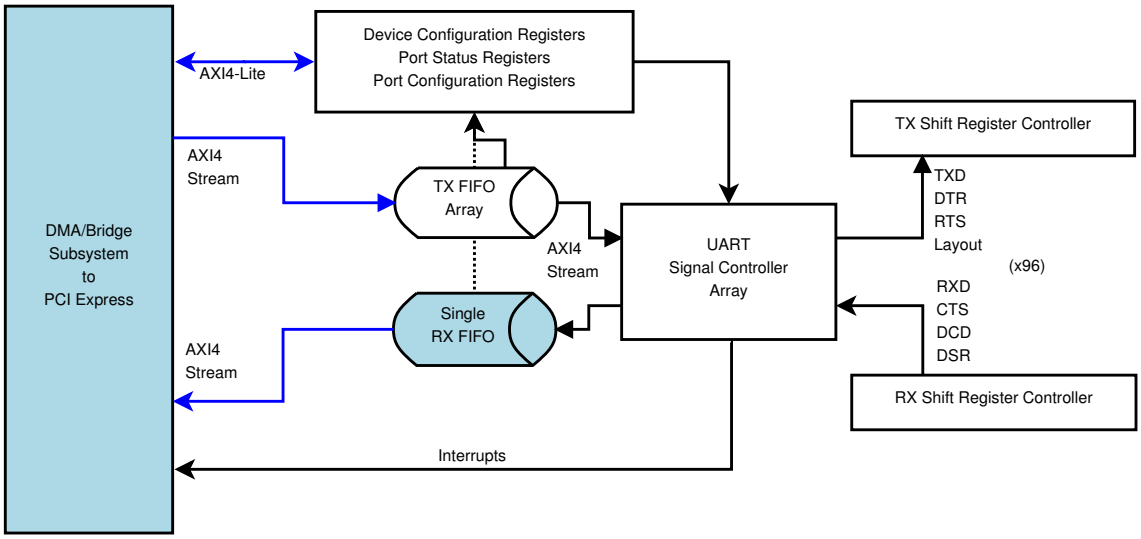



Figure 4.2: Block diagram of FPGA Design. Components highlighted in blue represent Xilinx IP cores, and blue signals represent interfaces where the Xilinx IP or FIFOs are used to cross between clock domains.

### 4.2.2 Transmitter FIFO

A typical single-input/single-output FIFO might be designed similarly to that shown in Figure 4.3. Data can be added to the FIFO by writing it at the cell marked *tail*, and read from the *head* cell. Every time data is added or removed, the respective head or tail pointer is moved by one position, or wraps around. The FIFO is full when *tail* is immediately behind *head*, and empty when they point to the same cell.

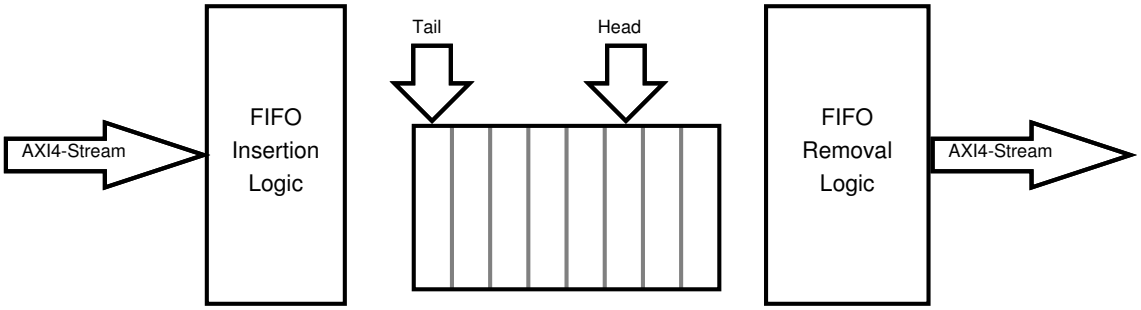



Figure 4.3: Design of a single FIFO.

Unfortunately, this does not lead to efficient use of memory when many small fifos are implemented on an FPGA. In an FPGA design, memory can either be made from flip-flops or from block RAMs. With an 8-bit datapath, 256-cell deep FIFOs, and 96 ports, this would require almost 200 000 flip-flops. For reference, the Xilinx XC7A200T used in this project only has 269 200 available, which must be shared with the DMA core and other UART logic. Using an individual block RAM per FIFO would also be wasteful, since block RAMs can only be allocated as small as 18 K b.

The solution is to emulate a series of FIFOs using a single block RAM. A diagram of

the TX FIFO array is shown in Figure 4.4. Memory for 8 FIFOs is allocated from a single FPGA Block RAM primitive. A table of head and tail *pointers* is stored in the Write Address and Read Address tables respectively. The FIFO insertion logic sorts data received and routes it to the appropriate FIFO, if there is space. The FIFO removal logic scans across each FIFO and transmits any available data to each of the UARTs (AXI4-Stream slaves).

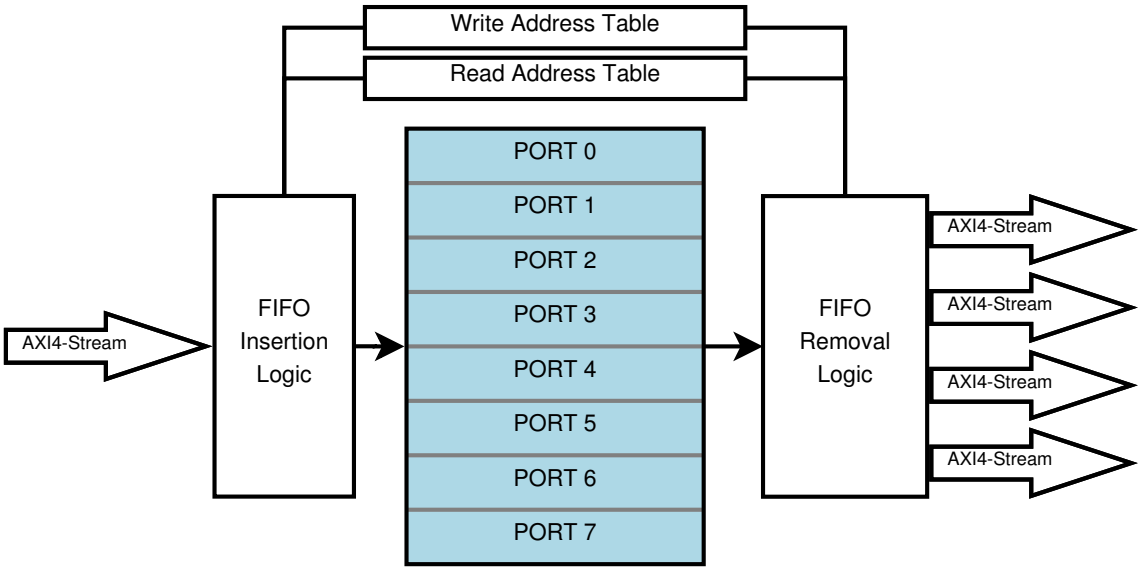



Figure 4.4: Architecture for the TX FIFO block. For brevity, only 4 outgoing AXI4-Stream interfaces are shown, however 8 feature in the block – one for each port in the Block RAM (blue). All of the components shown are synchronised with a single clock.

The FIFO array is repeated 12 times to make enough FIFOs for 96 ports. The *FIFO Insertion Logic* is able to accept a byte in a single clock cycle, so a 2-state state machine is used to route the data stream from the PCI Express core to each of the FIFO arrays, without introducing a bottleneck. A 4 K B FIFO is used to connect these FIFOs running at 73.728 MHz to the PCI Express core at 125 MHz.

### 4.2.3 Receiver FIFO

The DMA IP core uses an AXI4-Stream slave interface to read data. Since it is possible for data to be received while the DMA core is not running, it is prudent to insert a large FIFO before the DMA core, so that no data is lost. A single FIFO stores all data received by the 96-port UART, ready to be read by the DMA core. Individual streams of data from each port are combined into a single large stream with a very simple state machine.

This architecture has the added advantage that if some ports are used more than others, the total FIFO size is available to those ports – there is no wasted space in the FIFO. One potential disadvantage is that the driver is not able to decide which ports to receive data from, and in fact does not necessarily have knowledge of which ports are waiting to be

read. The inability to select which ports to receive is addressed using the ENABLE bit in the configuration registers (see Appendix A). The driver does not make use of this feature.

The RX signal chain is shown in Figure 4.5. A 64-bit FIFO was chosen because the DMA IP core requires a 64-bit datapath. Each UART also contains a 16 deep receiver FIFO, for local buffering and to generate flow control signals. The large FIFO depth of 128 K was largely arbitrary and chosen because of the sheer abundance of RAM on the FPGA used for prototyping. In testing, this FIFO never filled above 5 %, and could therefore have its size drastically reduced. The stream combiner is packs up to two characters, in the 32-bit format (see Table 4.1), into each FIFO cell.

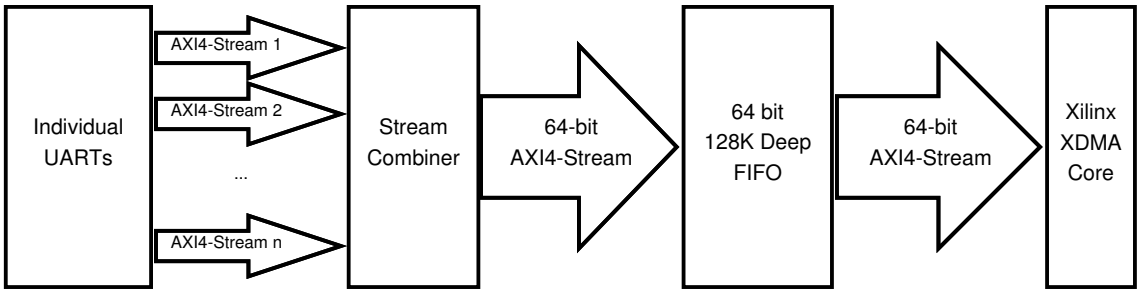



Figure 4.5: RX FIFO Signal Chain

The large FIFO is implemented using the Artix-7 built-in asynchronous FIFO primitives, which has the added advantage of being able to synchronise data from the UART clock domain (73.728 MHz) into the PCI Express clock domain (125 MHz).

### 4.2.4 UART Signal Controller Array

Each of the UARTs in the array shown in Figure 4.1 contains two AXI4-Stream interfaces for data, a 32-bit configuration/status register, and an RS-232 interface. These UARTs have full support for hardware DTR/DSR handshaking, RTS/CTS flow-control, and a small buffer for received data. The UART also has full support for a transparent hardware loop-back[2].

The DTR (Data Terminal Ready) signal is asserted low when the port is enabled and a data carrier (cable) has been detected – DCD (Data Carrier Detect) is asserted low. The user can override the DTR signal directly by explicitly enabling or disabling the port.

In hardware flow-control mode, the RTS (Request To Send) signal is always asserted low unless the small RX buffer is almost full. It can be overridden by the user once hardware flow-control has been disabled.

If loop-back is enabled,

- TXD is internally connected to RXD,
- DTR is internally connected to DSR,

[2]Transparent in the sense that the data being looped back is output on the physical TXD line, and routed into the RXD input. The receiver logic is not bypassed.

- RTS is internally connected to CTS, and
- DCD is internally asserted LOW.

### 4.2.5 Shift Registers and Controller

With 96 ports, each requiring eight signals[3], 768 I/O signals are required. FPGAs which have this many I/O pins available do exist, but they are prohibitively expensive. Because each input and output line does not need to be sampled at high speed, 8-bit CMOS shift registers were used to produce the extra I/O signals.

Shift registers on the PCB are grouped in pairs, and each pair of shift registers in turn creates a pair of UARTs. The configuration is shown in Figure 4.6. Each port has four input and four output signals, which were packed into 8-bit 74LV165A parallel-to-serial and 74LVC595A serial-to-parallel shift registers respectively.

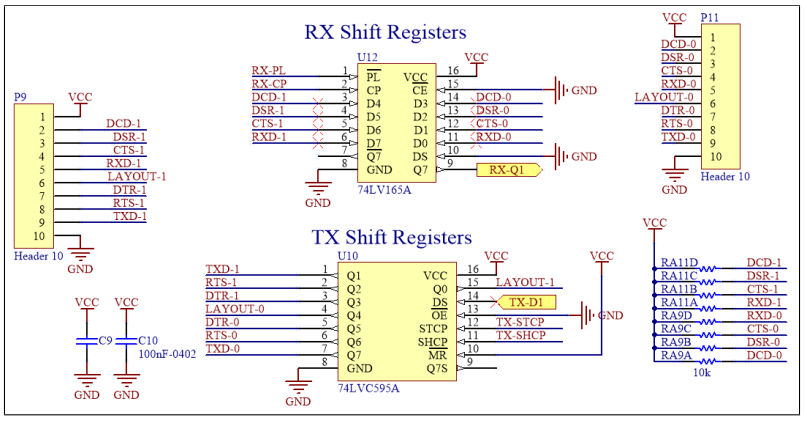



Figure 4.6: Shift register section schematic

Figure 4.2 shows that the FPGA design contains two modules for controlling the shift registers.

The controller for the *RX Shift Register* (74LV165A) is a state machine with 20 states, advancing on every second rising edge of the 73.728 MHz clock. The number of states was deliberately chosen so that the shift registers were sampled at 8 times the maximum baudrate supported. First the controller creates an active-low pulse on the parallel-load (RX-PL) line for the shift registers, then it repeatedly pulses the serial clock (RX-CP) line until all data has been shifted out.

The *TX Shift Register* (74LVC595A) controller operates in the opposite order. Data is shifted using the data pin and TX-SHCP[4]. This controller also operates on a 20-state state machine running on every second edge of the 73.728 MHz clock.

The two controllers operate on opposite clock edges, to avoid unnecessary *spikes* of current draw when too many outputs change at once. They also provide an entirely trans-

[3]RXD, TXD, RTS, CTS, DTR, DSR, CTS, and a general-purpose output used to select port layouts on Opengear console servers.

[4]shift clock

parent interface internally. The array of UART signal controllers is unable to distinguish between the shift registered I/O and actual I/O pins.

### 4.2.6 FPGA Simulation and Testbench

As part of the development process for the FPGA portion of the design, an extensive verification testbench was implemented. The testbench verifies all the functionality of the UART signal controllers and FIFOs by using incorporating VHDL models for the PCB shift registers and tracking each of the outputs. Xilinx-provided verification IP (VIP) cores are inserted on each of the AXI4-Lite and AXI4-Stream interfaces to ensure rigorous compliance.

The testbench operates in much the same way that the benchmarking suite does. It begins by setting each port individually into loop-back mode with hardware flow control. It then pushes twice as much data as will fit in any individual port's transmit FIFO[5] through a handful of the ports, one at a time. Once all of the data has been looped back from every port, the testbench pushes the same amount of data through every port at once.

All of the UART signal lines are emulated in the testbench by including VHDL models for the shift registers.

Each port has its TXD line monitored and checked against the input data set, and a single faulty output bit will cause the entire test to fail. Similarly, the AXI4-Stream interface which combines all of the received data is verified against the input data. The AXI4-Stream interface has its TREADY signal pulsed periodically, often enough that the flow control mechanism on each of the ports gets used. Any data dropped is interpreted as a data error, and causes the test to fail. The data supplied to each port is made unique by applying an exclusive-OR function with the destination port number to a pseudo-random input.

The testbench does not cover functionality relating to clock domain crossing, or PCI Express itself. Instead, the AXI/AXI-Stream interfaces to the IP cores which handle that functionality is checked. The testbench also only tests the UARTs at 230 400 baud, and never at slower baudrates. Likewise, directly entering data into the RXD line is not tested in the testbench.

## 4.3 Printed Circuit Boards

Two printed circuit boards (PCBs) were developed for this project. After initial estimates for the 96-port PCB suggested that it would draw an average of around 2 A at 2.5 V, an 8-port PCB with precise current measurement circuitry was designed. This was done so that the power consumption for the larger board could be estimated accurately, to help design the power supply for it.

[5]This is trivially parametrisable, and often tests were run with a reduced FIFO depth to make failures occur faster.

### 4.3.1 8-Port Test PCB

The 8-port test PCB has the same shift register configuration (see Figure 4.6) as the 96-port PCB. The only point of difference in the digital circuitry is that each of the signals controlling the shift registers are delivered individually, rather than as a large fan-out bus. This was done to allow tests where clocking signals were delivered out of phase with each other to reduce peak power consumption. Each pair of ports was also able to be disabled using a load switch. This board can be seen in Figure 4.7.

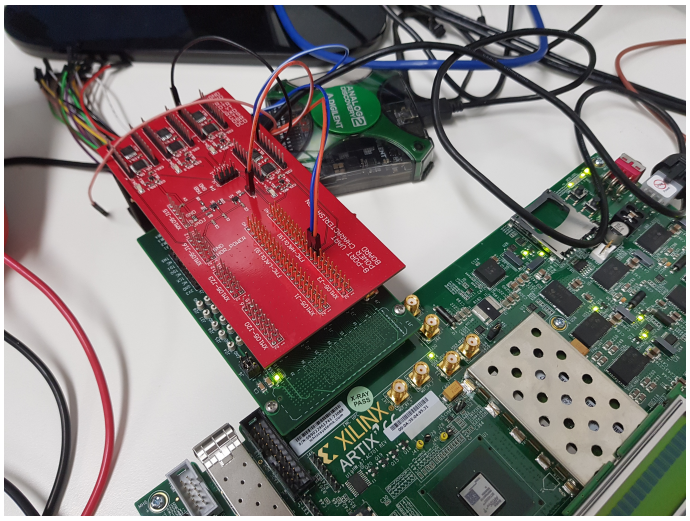



Figure 4.7: 8-port test PCB connected to the Xilinx AC701 development board via an XM105 FMC Debug breakout board.

Current delivered to the ports is measured by amplifying and measuring the voltage across a 1 Ω shunt resistor. The measurement circuit is shown in Figure 4.8. Unfortunately, this circuit was designed based on an inaccurate figure for current draw[6], and the actual current draw is not significant enough for U1 to output a meaningful voltage. Increasing the gain of U1 would have caused issues with bandwidth, and so the value of R7 would have needed to be increased to rectify the issue.

Oscilloscope measurements taken across R7 proved too unstable to be useful, and primitive measurements using a multimeter across R7 indicated that each pair of ports drew about 1.15 mA. Since the average current was measured to be so small, it was decided that the board would not be modified, and the higher frequency measurements were never taken.

### 4.3.2 96-Port PCB

A 96-port UART PCB has been design and manufactured. It utilises 48 74LV165A parallel-in/serial-out and 48 74LVC595A serial-in/parallel-out shift registers, both manufactured

[6]Initial estimates suggested that this design would be measuring currents around 40 mA to 200 mA.

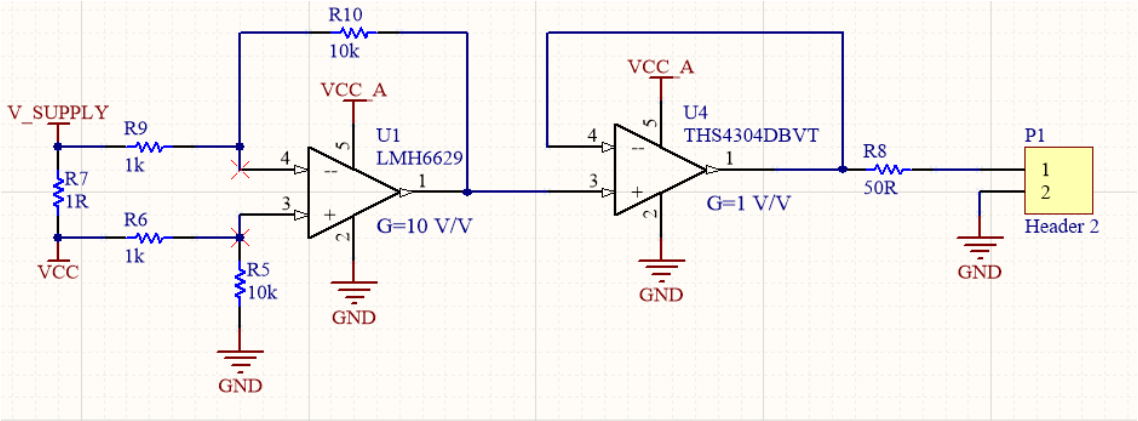



Figure 4.8: Current measuring analogue‘ front-end

by Nexperia. The PCB also features a Texas Instruments INA219 power supply monitor and TPS22917 integrated load switch.

Female headers on the board's underside connect via the XM105 breakout-board to the FPGA. Pairs of 9 header pins each contain all 8 RS-232 signals plus ground, at 2.5 V level. A pair of shift registers can be found between each pair of header pins. The schematic for this section is shown in Figure 4.6. Here, RX-PL, RX-CP, RX-CE, TX-OE, TX-STCP, and TX-SHCP are shared control signals common to each port, and RX-Q and TX-D are data signals unique to each pair of registers.

The final board can be seen in Figure 4.9.

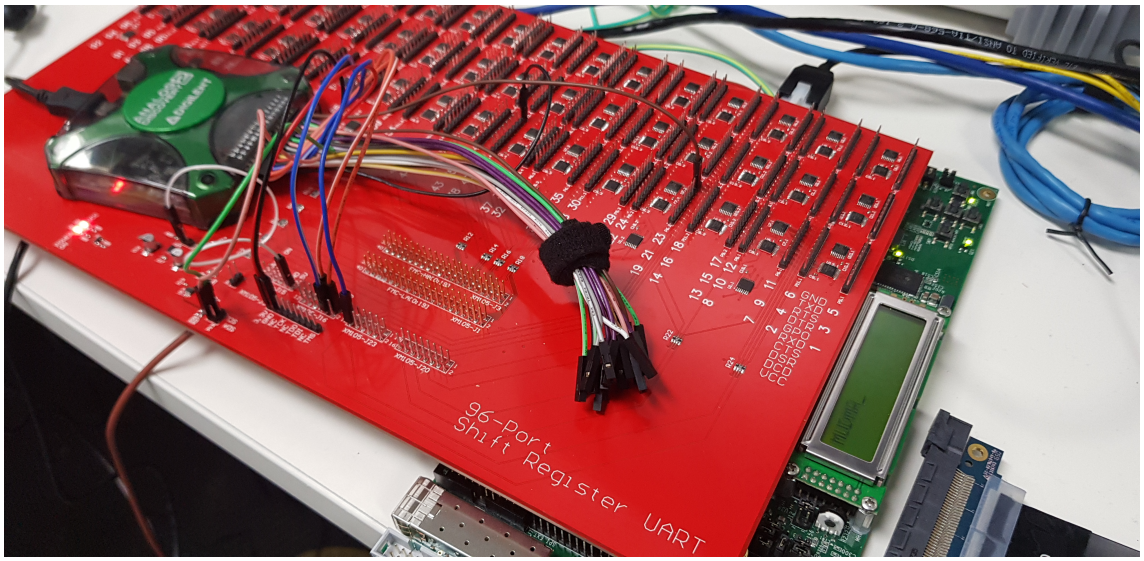



Figure 4.9: 96 Port PCB

### 4.3.3 DC Power Budget

A power budget was produced for the first revision of the 96-port board, which suggested that it would use in about 5 W. This lead to the creation of the 8-port board and was

found to be an inaccurate figure. Direct measurements using the 8-port board gave a new estimate of 250 mW to 300 mW for the 96-port board.

The power budget was developed assuming the worst-case conditions for the device: running at the maximum clock frequency and driving large capacitive loads at high temperature. Once most of these assumptions were scaled back and made more practical, the revised estimate for power draw came to about 1.13 W total for 96 ports. The new estimate maintains the assumption that every input and output signal can switch on every clock cycle, despite being impossible in the design. For this reason, the new figure was considered to be an acceptable over-estimation.

This PCB draws about 400 mW at 2.5 V, which is significantly less than the revised estimate. Most importantly, the PCB draws significantly less than both the 10 W the AC701 is capable of supplying, and the 1.13 W the design allowed for.

The Jupyter notebook used to perform the calculations can be found in Appendix D.

## 4.4 Linux Driver

The Linux driver serves two purposes:

- It must allow user applications to read and modify status and configuration information about the hardware, and
- present a simple interface for transmitting and receiving data from each of the UARTs on the FPGA.

### 4.4.1 Design Overview

Ordinarily, the Linux TTY subsystem would be used to create a unified interface for applications and driver would need to implement a hardware-specific back-end for it. However, using a character device for each of those functions is sufficient to demonstrate the functionality and performance goals for the project. As such, two character devices have been created: `mucfg` and `mudata`.

The character device `mucfg` is mounted at `/dev/mucfg`, and supports `mmap` accesses only – the `read` and `write` system calls are not implemented. By accessing memory mapped with `mmap`, a userspace application is able to access the configuration registers for each of the UARTs, as well as global configuration data. Userspace utilities have been developed to manipulate this character device, see Section 4.6.

The other character device, `mudata` is created with enough *minor* device numbers to represent each UART port individually. One filesystem node for each port is created at `/dev/mudata0`, `/dev/mudata1`, and so on. By reading from the FPGA ROM at start-up, the driver automatically determines how many ports are present, and creates the appropriate number of filesystem nodes. Userspace applications are able to write to and

read from each port using the `read` and `write` system calls. When a port is in loop-back mode, its corresponding character device is almost indistinguishable from a pipe into `cat`.[7]

The functionality behind `mudata` can be broken into two independent sections – the transmitter and receiver chains. These are outlined in sections 4.4.2 and 4.4.3 respectively.

### 4.4.2 Transmitter Chain

When the user initiates a write to `mudata`, the driver creates a buffer of packets from each byte received using the packet format shown in Table 4.1. This buffer is then added to a queue of outgoing buffers waiting to be transferred onto the device, along with a `waitqueue` and a reference counter for the buffer. A task to perform the DMA transfer is then scheduled to run. Since writing to a full FIFO on the FPGA would bring the device to a crawl, writes are limited to the depth of the transmission FIFOs on the hardware.

If the user requested blocking I/O, the driver then waits on the included `waitqueue` for a notification to signify that the transfer completed successfully and frees the buffer. The reference counter ensures either the interrupt handler or the writing thread will always free the buffer, but never both.

The queue of outgoing buffers is implemented using an array of linked lists – one for each port. The Linux kernel has a built-in circular doubly-linked list implementation, and that is used for each of these lists. By using a circular list, there is no need to worry about wasting time traversing the list to insert a buffer at the end or retrieving from the start – all of the relevant operations on these lists can be executed in $O(1)$ time. Being able to quickly insert and remove buffers is a major consideration since during a benchmark, each port will process about 4000 buffers[8].

Whenever an outgoing DMA transfer is requested, the driver scans through all of the active ports on the UART, and determines how many spaces are available in each of the outgoing FIFOs. For each port with space left in its FIFO, the first available buffer from the user is selected to be transferred. The order in which FIFOs are filled is determined by the number of available spaces in each FIFO – the FIFOs with the most available space get filled first. This minimises the amount of time each FIFO spends empty.

The DMA transfer task creates a scatter-gather list from the selection of outgoing buffers, and uses the DMA kernel API to map them onto the PCI Express bus address space. The scatter-gather list is then converted to a bus-mapped linked list of XDMA descriptors (see Section 2.10.1 and [23]). The Xilinx DMA module is configured using I/O memory writes to the device. An interrupt request from the XDMA module signals that the transfer is complete.

When the *transfer complete* interrupt request is received, any write threads waiting for

[7] On most modern Linux systems, `cat` can move copy data at rates exceeding 230 400 baud, and may involve less copying operations.

[8] 1 MB is sent per port, limited to 256 bytes per transfer

buffers in that transfer are *woken up*[9]. Each buffer is then freed. A race condition with the write thread, which would potentially cause a double-free error or leaked memory, is avoided using the atomic reference counter mentioned above.

### 4.4.3 Receiver Chain

The receiver chain effectively works in reverse to the transmitter chain. The driver maintains a FIFO for each port and whenever the `read` syscall is used, information is first served from this FIFO. If the FIFO is not sufficiently deep so as to serve the read request completely, a transfer from the FPGA is scheduled. If no data was able to be served at all, and the user application did not specify a non-blocking read, then the driver will wait to be *woken up* by the end of the transfer operation. When possible, the driver will always return some data, even if the user requested more than was available, irrespective of the non-blocking option.

Whenever one of the `mudata` endpoints requests additional data, the driver reads the FPGA's receiver FIFO count register and initiates a transfer for all data in the FIFO. When data is to be read from the FPGA, a single 128 KB buffer is used. This buffer is allocated when the FPGA's PCI Express endpoint is detected. This simplifies memory management, as there is no need to repeatedly allocate, map, unmap, and free transfer buffers.

When an interrupt signalling the end of the transfer is complete arrives, the contents of the buffer are sorted into each of the internal port FIFOs. Any data which cannot be placed in the correct FIFO is dropped and causes a warning message to be printed to the kernel log. Similar to the transmitter chain, any threads waiting on data will be *woken up*.

The FIFO for each port has a depth of 128 KB. This was chosen to match the FPGA RX FIFO (see Figure 4.5) so that no single port could possibly lock-up the device.

## 4.5 Changes in response to benchmarking

In response to benchmarking the new UART, several performance enhancements were made to both the FPGA and its driver.

Initially, the FPGA ran at 36.864 MHz, a large multiple of the top baud-rate of 230 400 baud. It was found that doubling this to 73.728 MHz meant that the FIFOs were able to accept data faster. This also meant that the driver did not have to break up buffers and shuffle them to improve performance. If the driver *did* break up the outgoing buffers, there might still be an improvement in performance, but this much simpler change makes that unnecessary.

A small FIFO was added to the input of all the transmission FIFOs so that the Xilinx DMA would perceive the transfer as being complete earlier, and could send the appropriate

[9] This is not a colloquialism – the API for this operation is called `wake_up(...)`

interrupt faster. This reduced the latency between outgoing transfers and marginally improved performance.

## 4.6 Userspace Utilities

Two userspace utilities have been built in order to assist in both FPGA and driver development: `lsport` and `portset`. Both of these utilities access the configuration memory space in the FPGA device and report readings to the user. By default, both tools access this memory space using the `/dev/mucfg` char device, however they can access the memory directly using the PCI Express device' `sysfs` entry.

The first tool, `lsport`, reads the configuration memory and displays key information about each of the ports in the multi-port UART. Specifically, `lsport` displays

- the number of ports configured in the device,
- counters for both correct and erroneous streaming writes,
- the status of the DMA controller,
- the amount of data present in the receiver FIFO, and
- the configuration settings for each of the ports configured.

The other tool, `portset`, allows a user to configure the ports attached to the device by changing the baudrate, byte width, parity, and loop-back settings. This tool performs the same functionality that a utility such as `stty` might perform, without the driver having to implement the relevant `ioctl`s and TTY interface.

# Section 5

# Benchmarking Results

What follows is an analysis of the results obtained through benchmarking the FPGA UART against the Opengear CM7196.

## 5.1 Sample of Benchmarking Data

The FPGA design and driver were benchmarked using three different CPU configurations.

1. Intel quad-core i7-960 CPU at 2.8 GHz
2. Intel i7-960 running with single-core only at 1.6 GHz
3. The same as 2, while under an intense computational load.[1].

The CM7196 console server's mainboard does not include any standard PCI Express connectors. As such it was not possible to directly connect the FPGA development kit to its CPU. Benchmarks involving the ARM CPU were attempted using the mainboard for a similar Opengear product, the IM7248, however incompatibilities between the PCI Express connector on the IM7248 mainboard and AC701 development kit[2] made it infeasible to test this configuration.

The benchmarking data shown here correspond to tests at a baudrate of 230 400 baud. A comprehensive set of testing data across all relevant hardware can be found Appendix B.

## 5.2 Comparison with CM7196

Benchmarking data for the Opengear CM7196 is provided in Figures 2.3 and 2.4. As mentioned in Section 2.1, the CM7196 performance at 230 400 baud declines sharply when more than 8 ports are used at once. A similar trend can be seen when the same device drives more than 16 ports at 115 200 baud. In both of these scenarios, the device appears to reach a maximum total bandwidth of around 190 kB/s to 200 kB/s.

[1] 16 threads each running bogo-sort on an array of 10 million pseudo-random integers

[2] The IM7248 uses the standard 4-lane PCI Express edge connector, however it only connects three of the lanes. The lack of support for MSI interrupts on the IM7248 also caused issues for the Xilinx DMA IP core.

When the device bandwidth is past this saturation point, the CPU utilisation reaches 100 %. This indicates that the performance for the CM7196 is somehow bound by the CPU.

Figure 5.1a shows that the new FPGA solution, when connected to a down-clocked x86 CPU does exhibit such behaviour. As the number of ports increases, the average data rate per port decreases only marginally. When 96 ports are in use, the average data rate per port is roughly equal to that of the CM7196 with one active port. The degradation in performance seen is likely caused by the slight delay between transfers. Under benchmark conditions when all 96 ports are being used at once, the procedure for sending data is roughly as follows:

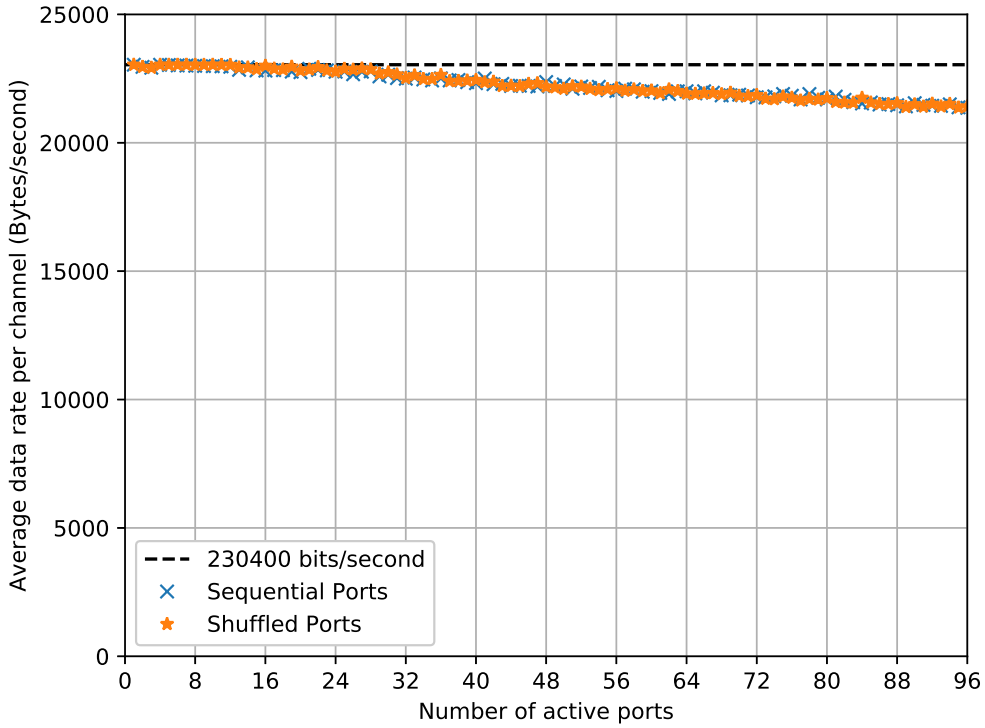
1. Driver starts with several pre-packed outgoing buffers
2. Driver queries all of the ports and generates a list of available spaces in the transmission FIFOs, and sorts that list.
3. In order of most to least urgent, the driver selects one buffer per port and maps it to bus memory
4. The driver uses memory mapped I/O to launch a transfer, and waits for the interrupt to signal its completion.
5. After receiving the interrupt, the driver frees the mapped and allocated memory and requests another outgoing transfer.

At 230 400 baud with 256 B FIFOs, it takes roughly 11 ms to empty a FIFO. By the time all of the FIFOs have been written to, the latency between step 4 finishing and the next transfer starting causes some of the outgoing FIFOs to become empty before being completely refilled again. This explains the slight degradation in performance as more ports are added.

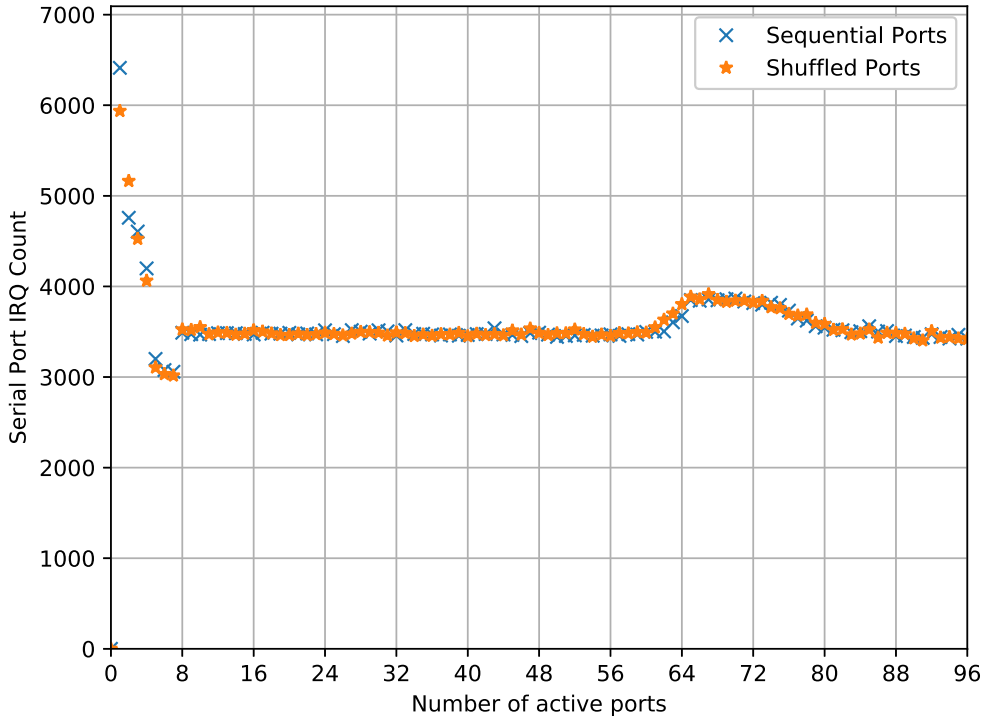
Figure 5.1b also shows that during each test above 8 ports the CPU only receives around 3500 to 4000 interrupts, and this value is largely independent of the number of ports being tested. For comparison, the CM7196 receives over 100 000 interrupts when more than half the ports are being used. This improvement over the CM7196 is the result of a few key architectural decisions:

- Only a single receive FIFO is used, and so there is only a single interrupt which fires when there is data waiting to be read.
- When an interrupt is received, the driver masks whichever interrupt fired until the *bottom half* function has finished processing it.
- If user applications have made enough data available, the driver is able to completely fill every port's transmission FIFO with a single interrupt firing.

The combination of these three architectural features is what makes the developed solution significantly faster than the original solution.




(a) Average data-rate per port




(b) Interrupt Request (IRQ) Count

Figure 5.1: Benchmarking data for the FPGA connected to an Intel single-core i7-960 x86 CPU at 1.6 GHz

# Section 6

# Conclusions

This project has delivered real value to Opengear by providing

- a complete benchmarking methodology,
- a complete set of benchmarking data for their current products,
- an alternative solution to the current UART system used in their console servers and products.

Even if an FPGA-based solution is not adapted for use in a production environment, the benchmarking methodology and suite can now be used to evaluate any future designs involving multi-port UARTs. The alternative solution is valuable in the event that Exar were to cease production of PCI Express multi-port UARTs, and a high performance replacement was needed quickly.

## 6.1 Future Work

The objective of this project was to produce a *proof of concept* for an alternative, high-performance PCI Express UART using an FPGA. Now that is complete, there are many natural directions for the project. These are divided into performance enhancements and usability enhancements.

### 6.1.1 Performance Enhancements

The register file central to the device is only accessible using memory-mapped I/O to an AXI4-Lite endpoint. Typically, the driver will read the large sections of the register file at once, which could prove to be expensive performance-wise. If necessary, using DMA to transfer the entire register file at once may help to improve performance.

Currently, data is copied from userspace and processed before being sent via DMA to the FPGA. If the communication protocol was changed such that metadata (destination UART, etc.) did not get embedded directly in the data, it would not need to be processed by the kernel, and zero-copy transfers could be implemented. This way, buffers are mapped

to bus memory directly from userspace and never get copied by software, for a potential performance improvement.

Initiating a transfer via DMA requires mapping kernel memory pages to the bus address space, manipulating the MMU, programming the DMA hardware via memory-mapped writes, and waiting for interrupts. This is a large overhead in the case where only very short transfers are being undertaken. Adding a bypass for the DMA system could potentially improve performance for small transfers.

The Xilinx *DMA/Bridge Subsystem for PCI Express* IP core favours streaming transfers between the bus and the FPGA. If the FPGA were able to use directly fill FIFOs on the bus, then a lot of the time spent by the CPU manipulating the DMA engine, and potentially a lot of on-chip resources, could be spared. This architecture could potentially be realised using the *AXI Memory Mapped to PCI Express* IP core.

### 6.1.2 Usability Enhancements

Currently, the driver creates two character devices and exposes the FPGA's configuration memory space to user applications directly. A typical UART driver for Linux should interface with the TTY subsystem, so that line and MODEM control settings can be changed using the standard `termios` interface. This incompatibility has meant that the `/dev/mudata` devices are not compatible with `minicom`, `screen`, and other tools expecting a TTY interface. Implementing the standard UART interface will fix this incompatibility.

The PCB produced has no support for RS-232 signal levels, and instead uses 2.5 V signalling. The addition of line drivers is a definite requirement for moving the design into production.

Provisions have been made in the 32-bit data transfer format (see Table 4.1) to signal break conditions, XON/XOFF generation, and parity errors, but these are not actually implemented. Currently, the only way for a UART to signal any kind of error is by setting changing the error bit in the status register, however the driver has no way to reset this bit or to determine what the error was.

In the current design for 96 ports, a single large FPGA uses 96 shift registers to generate all of the required I/O signals. For both of these reasons, the new design is expensive, certainly more so than the original. Additionally, the vast majority of the console servers Opengear produces are significantly smaller than 96 ports – some as few as eight. It is not sensible to have a single design for 96 ports and only partly use it, where instead it would be more practical to build a 16-port or 24-port UART using smaller FPGAs. In such a case, having less ports per FPGA may eliminate the need for shift registers.

Currently, legacy-style interrupts are not supported by the hardware and may cause the DMA core to crash. This is caused by an incompatibility with the interrupt handling module, wherein a race-condition causes interrupts to be de-asserted early.

# Section 7

# Professional Development

## 7.1 Key Learning

Before starting this project, I had minimal experience developing modules for the Linux kernel, and only academic experience using an FPGA. From a technical standpoint, I have now gained much experience in both those areas. This project has required that I design hardware which uses professional IP to interface with the technologies (AXI, PCI Express, DMA, RS-232, etc.) which is used in regular desktop computers and other high performance digital systems. As a result of what I have learned, I now feel far more confident in my ability to design and implement complex digital systems.

Aside from all the technical knowledge and experience I have gained in this project, I have become more familiar with how a large and complex project is managed. None of the projects I worked on at previous workplaces have come close to the scale of this project, and managing all of this extra complexity was a new challenge for me.

Initially, I jumped straight into building different parts the project, and I had a lot of ideas about how I would build them, without any real consideration for how they would work together. This actually lead me to explore a number of the options and possible design decisions that were available to me.

This pattern of briefly jumping straight in to become familiar with the solution space and quickly establish what would and would not work before then planning the end-product is how most of the *investigation* work was done. As I become more fluent in a problem domain, this initial *jump-in* becomes far less necessary. An example of when I applied this pattern was mentioned in reflection 3 – 15th April. I developed a series of trivial kernel modules which each contained parts of the functionality I needed to include in the final driver. Eventually, most of these smaller modules got merged into a single module, which formed the basis of the Linux driver.

Something I found frustrating in the latter half of the project was that the automatic testbench for the FPGA design failed to catch many of the minor bugs prevalent in the design. Including extra test cases to cover bugs previously missed encouraged me to think

about vulnerabilities within the design, and even more about how different components can fail. This work debugging my solution provides incredibly valuable experience which will help me in future work.

The driver had full functionality by about week 21 of the project. The scope did not include implementing the standard TTY interface, alas everything which could normally be achieved using that interface was instead made possible through character devices. Unfortunately, by this point the driver was incredibly slow, and when run using the quad-core Intel i7 at full speed actually displayed the same performance as the original Exar UARTs with the single-core ARM CPU. This meant that I had to re-think and refactor much of how the driver worked.

My participation in this project has placed me in a high-tech design environment. By surrounding myself with engineers who design computer hardware for a living, it has changed the notion that serious hardware and computer engineering is something out of reach as a viable career path.

## 7.2 Engineering Australia Competencies

As demonstrated in Section 7.1 and in the monthly reflections (Appendix C), I believe that I have developed the following Engineering Australia Stage One competencies:

**1.1** *Comprehensive, theory based understanding of the underpinning natural and physical sciences and the engineering fundamentals applicable to the engineering discipline.*

**1.2** *Conceptual understanding of the mathematics, numerical analysis, statistics, and computer and information sciences which underpin the engineering discipline.*

**1.3** *In-depth understanding of specialist bodies of knowledge within the engineering discipline.*

**1.4** *Discernment of knowledge development and research directions within the engineering discipline.*

**2.1** *Application of established engineering methods to complex engineering problem solving.*

**2.2** *Fluent application of engineering techniques, tools, and resources.*

**2.3** *Application of systematic engineering synthesis and design processes*

**2.4** *Application of systematic approaches to the conduct and management of engineering projects.*

**3.1** *Ethical conduct and professional accountability.*

**3.2** *Effective oral and written communication in professional and lay domains.*

**3.3** *Creative, innovative, and pro-active demeanour.*

**3.5** *Orderly management of self, and professional conduct.*

# Bibliography


[1] Exar Corporation, *XR17V358 high performance octal PCI Express UART*, Apr. 2015. [Online]. Available: https://www.exar.com/ds/xr17v358_105_042215.pdf (visited on Feb. 21, 2018).

[2] V. S. P. Nayak, G. K. Saitejdeep, L. R. Kumar, N. Ramchander, and K. Madhukar, "Modular approach for customizable uart", in *2016 IEEE International Conference on Recent Trends in Electronics, Information Communication Technology (RTEICT)*, May 2016, pp. 748–750. DOI: 10.1109/RTEICT.2016.7807925.

[3] Texas Instruments, *PC16550 universal asynchronous receiver/transceiver with FIFOs*, May 2015. [Online]. Available: http://www.ti.com/lit/ds/symlink/pc16550d.pdf (visited on Jan. 16, 2018).

[4] Q. Yi, M. Shi, and S. Li, "Design of USB-UART interface converter and its FPGA implementation", in *2017 IEEE 2nd Advanced Information Technology, Electronic and Automation Control Conference (IAEAC)*, Mar. 2017, pp. 1399–1403. DOI: 10.1109/IAEAC.2017.8054244.

[5] Oxford Semiconductor, *OXPCIe954 PCI Express bridge to quad serial port*, Feb. 2008. [Online]. Available: https://www.semiconductorstore.com/pdf/newsite/oxford/OXPCIe954_ds.pdf (visited on Mar. 1, 2018).

[6] PLX Technology, *Product discontinuance notice 2013-8*, Dec. 2013. [Online]. Available: https://www.mouser.com/PCN/PLX_Technology_2013_8.pdf (visited on Mar. 1, 2018).

[7] Oxford Semiconductor, *OXPCIe958 PCI Express bridge to octal serial port*, Feb. 2008. [Online]. Available: https://www.semiconductorstore.com/pdf/newsite/oxford/OXPCIe958_ds.pdf (visited on Jun. 21, 2018).

[8] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers*, 3rd ed. United States of America: O'Reilly Media, Feb. 2005, ISBN: 0-596-00590-3.

[9] National Instruments, *PCI Express - an overview of the PCI Express standard*, Nov. 5, 2014. [Online]. Available: http://www.ni.com/white-paper/3767/en/ (visited on Mar. 7, 2018).

[10] Altera, *Cyclone V hard IP for PCI Express user guide*, Jun. 2012. [Online]. Available: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/ug/archives/ug-01110-1.2.pdf.

[11] C. Maxfield, *The Design Warrior's Guide to FPGAs*. United States of America: Newnes, 2004, ISBN: 0-7506-7604-3.

[12] A. Moore, *FPGAs for Dummies*, 2nd Intel Special Edition, R. Wilson, Ed. United States of America: John Wiley & Sons, Inc., 2017, ISBN: 978-1-119-39049-7.

[13] Xilinx Corporation, *7 series FPGAs configurable logic block user guide*, Dec. 27, 2016. [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf.

[14] Xilinx Corporation, *7 series FPGAs integrated block for PCI Express v3.3*, Oct. 4, 2017. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/pcie_7x/v3_3/pg054-7series-pcie.pdf.

[15] Xilinx Corporation, *Vivado design suite - HLx editions*. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado.html (visited on Jun. 21, 2018).

[16] Intel, *Intel FPGA development tools*. [Online]. Available: https://www.altera.com/products/design-software/overview.html (visited on Jun. 21, 2018).

[17] Lattice Semiconductor, *Design software and ip*. [Online]. Available: http://www.latticesemi.com/software (visited on Jun. 21, 2018).

[18] Intel Newsroom, "Intel completes acquisition of Altera", Dec. 28, 2015. [Online]. Available: https://newsroom.intel.com/news-releases/intel-completes-acquisition-of-altera/ (visited on Mar. 8, 2018).

[19] Intel, *Avalon interface specifications*, May 8, 2017. [Online]. Available: https://www.altera.com/en_US/pdfs/literature/manual/mnl_avalon_spec.pdf.

[20] Xilinx Corporation, *AXI reference guide*, Nov. 15, 2012. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug761_axi_reference_guide.pdf.

[21] ARM, *AMBA4 AXI4-Stream Protocol*, version 1.0, Mar. 3, 2010. [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0051a/index.html (visited on Mar. 21, 2018).

[22] ARM, *AMBA AXI and ACE protocol specification*.

[23] Xilinx Corporation, *DMA/bridge subsystem for PCI Express v4.0*, Dec. 20, 2017. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/xdma/v4_0/pg195-pcie-dma.pdf.

[24] Xilinx Corporation, *AXI memory mapped to PCI Express (PCIe) gen2 v2.8*, Oct. 4, 2017. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/axi_pcie/v2_8/pg055-axi-bridge-pcie.pdf.

[25] Xilinx Corporation, *AXI UART 16550 v2.0*, Oct. 5, 2017. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/axi_uart16550/v2_0/pg143-axi-uart16550.pdf.

[26] A. Sarwar, "CMOS power consumption and Cpd calculation", Texas Instruments, Jun. 1997. [Online]. Available: http://www.ti.com/lit/an/scaa035b/scaa035b.pdf (visited on Mar. 13, 2018).

[27] S. Radu, R. E. DuBroff, T. H. Hubing, and T. P. V. Doren, "Designing power bus decoupling for cmos devices", in *1998 IEEE EMC Symposium. International Symposium on Electromagnetic Compatibility. Symposium Record (Cat. No.98CH36253)*, vol. 1, Aug. 1998, 375–380 vol.1. DOI: 10.1109/ISEMC.1998.750120.

[28] H. J. M. Veendrick, "Short-circuit dissipation of static CMOS circuitry and its impact on the design of buffer circuits", *IEEE Journal of Solid-State Circuits*, vol. 19, no. 4, pp. 468–473, Aug. 1984, ISSN: 0018-9200. DOI: 10.1109/JSSC.1984.1052168.

[29] H. Kavianipour and C. Bohm, "High performance FPGA-based scatter/gather DMA interface for PCIe", in *2012 IEEE Nuclear Science Symposium and Medical Imaging Conference Record (NSS/MIC)*, Oct. 2012, pp. 1517–1520. DOI: 10.1109/NSSMIC.2012.6551364.

[30] Xilinx Corporation, *AC701 evaluation board for the Artix-7 FPGA*, Apr. 7, 2015. [Online]. Available: https://www.xilinx.com/support/documentation/boards_and_kits/ac701/ug952-ac701-a7-eval-bd.pdf.

# Appendix A

# Configuration and Status Register Layout

The layout for each port's status/configuration register is shown in Figure A.1. A mode of RO indicates that a field is read-only, and unaffected by writes. Fields marked as '—' are unused, and their value is both undefined and meaningless.

Table A.1: Configuration and Status register layout

| Bits | Description | Mode |
|---|---|---|
| 31 | RX Error detected | RO |
| 30 | Reserved | — |
| 29 | DCD | RO |
| 28 | TX FIFO almost empty | RO |
| 27 | TX FIFO empty | RO |
| 26 | CTS | RO |
| 25 | DSR | RO |
| 24 | TF FIFO almost full | RO |
| 23 | Reserved | – |
| 22 | Reserved | – |
| 21 | RX Error interrupt enable *(Not implemented)* | RW |
| 20 | TX FIFO almost empty interrupt enable | RW |
| 19 | TX FIFO empty interrupt enable | RW |
| 18 | Ignore DTR | RW |
| 17 | Loopback enable | RW |
| 16 | RTS override | RW |
| 15 | Reserved | – |
| 14 | General Purpose Output (GPO) | RW |

Table A.1 – continued from previous page

| Bits | Description | | | Mode |
|---|---|---|---|---|
| 13<br>12 | Stopbit Mode – | 00 - 1 stopbit<br>10 - 2 stopbits | 01 - 1.5 stopbits<br>11 - reserved | RW |
| 11<br>10 | Parity Mode – | 00 - Disabled<br>10 - Even parity | 01 - Odd parity<br>11 - Stick parity | RW |
| 9<br>8<br>7 | Databits – *see below for possible values* | | | RW |
| 6<br>5<br>4<br>3<br>2 | Baud rate – *see below for possible values* | | | RW |
| 1 | Hardware flow control enable | | | RW |
| 0 | Port enable | | | RW |

**Supported values for databits (bits 9-7)**

| | | | |
|---|---|---|---|
| 000 - 5 bits | 001 - 6 bits | 010 - 7 bits | 011 - 8 bits |
| | | | others - Reserved |

**Supported values for baud rate (bits 6-2)**

| | | | |
|---|---|---|---|
| 00000 - 50 | 00001 - 75 | 00010 - 110 | 00011 - 134 |
| 00100 - 150 | 00101 - 200 | 00110 - 300 | 00111 - 600 |
| 01000 - 1200 | 01001 - 1800 | 01010 - 2400 | 01011 - 4800 |
| 01100 - 9600 | 01101 - 19200 | 01110 - 38400 | 01111 - 57600 |
| 10000 - 115200 | 10001 - 230400 | | others - Reserved |

# Appendix B

# All Benchmarking Data

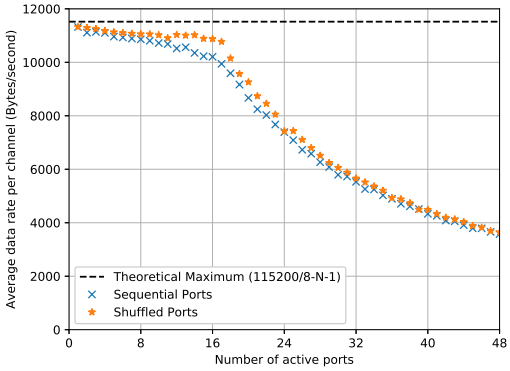
What follows is a comprehensive set of benchmarking data. These benchmarks were carried out as per Section 3.5, in the following configurations:

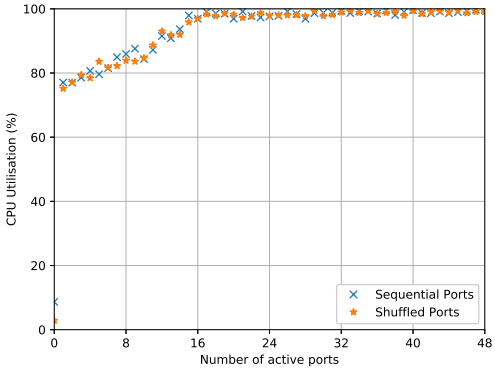
- Opengear IM7248 – 48 Port Console Server
  - 115 200 baud
- Opengear CM7196 – 96 Port Console Server (see Section 2.1)
  - 230 400 baud
  - 115 200 baud
- FPGA + Intel i7-960, quad-core at 2.8 GHz
  - 230 400 baud
  - 115 200 baud
- FPGA + Intel i7-960, single-core at 1.6 GHz
  - 230 400 baud
  - 115 200 baud
  - 38 400 baud
- FPGA + Intel i7-960, single-core at 1.6 GHz, artificially high CPU load
  - 230 400 baud
  - 115 200 baud

In all tests, the UARTs are configured with 8 data bits, 1 stop bit, no parity, and RTS/CTS flow control enabled.

## B.1 Opengear IM7248

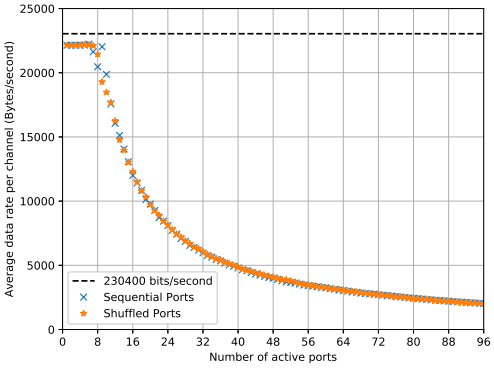
115 200 baud

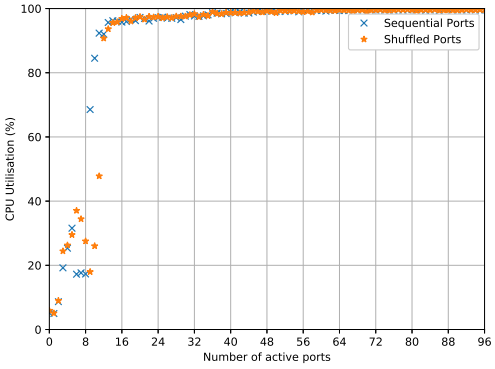


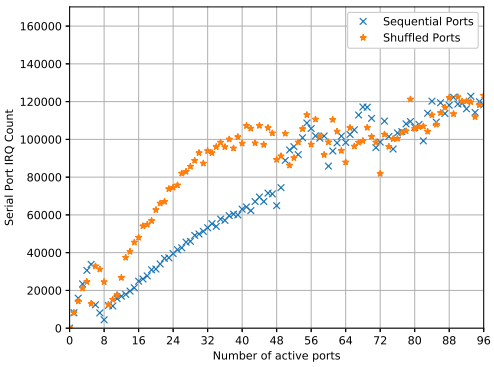




## B.2 Opengear CM7196

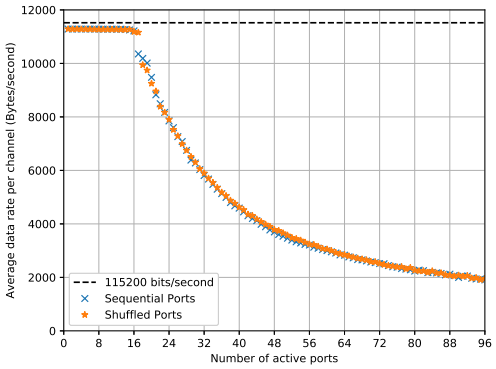
230 400 baud

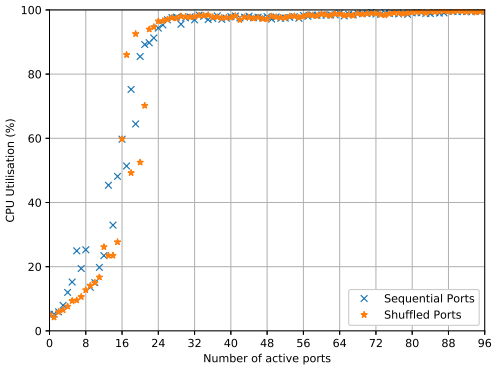


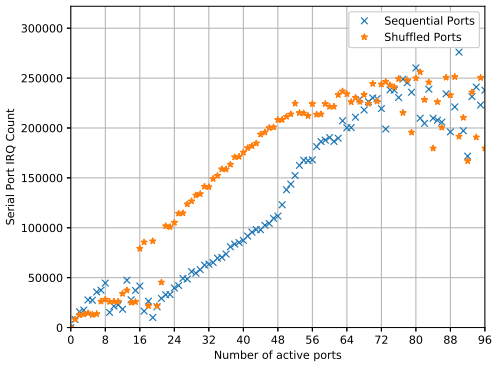



115 200 baud



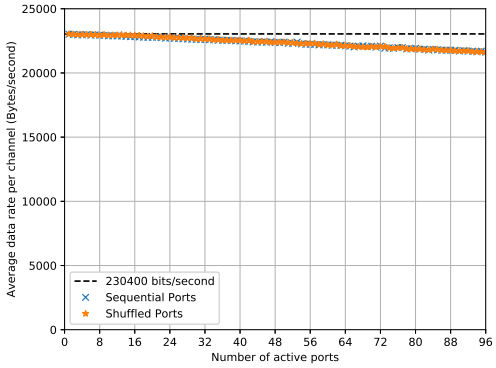



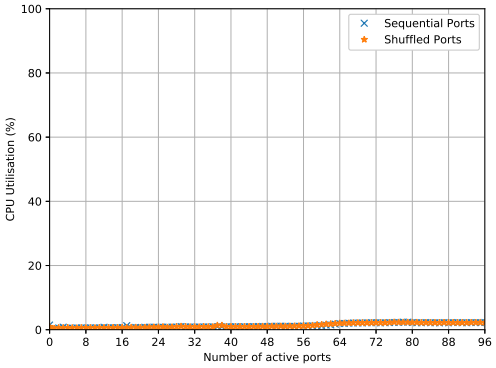



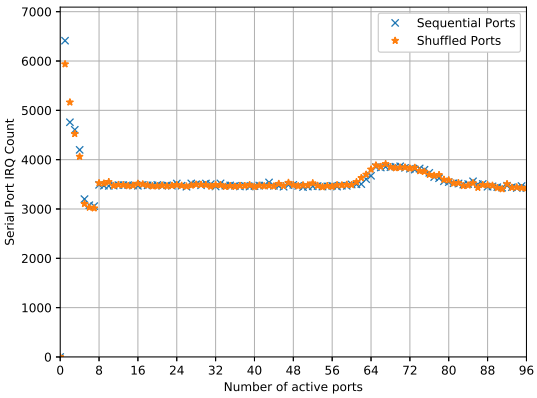

## B.3 Intel i7-960, Quad Core

CPU utilisation statistics for the quad-core configuration are not normalised with the core count. Also, this CPU supports hyper-threading with two threads per core. Thus, a reported 12.5 % utilisation actually represents 1 thread at 100 %.

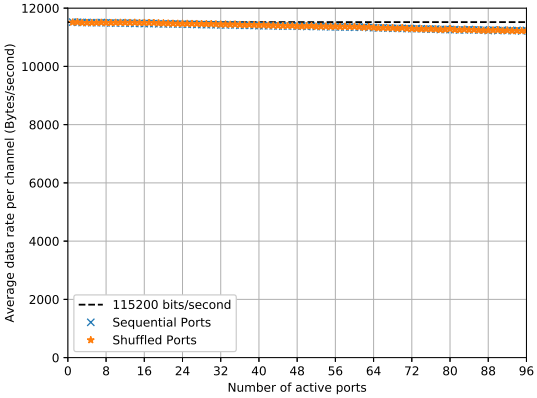
230 400 baud

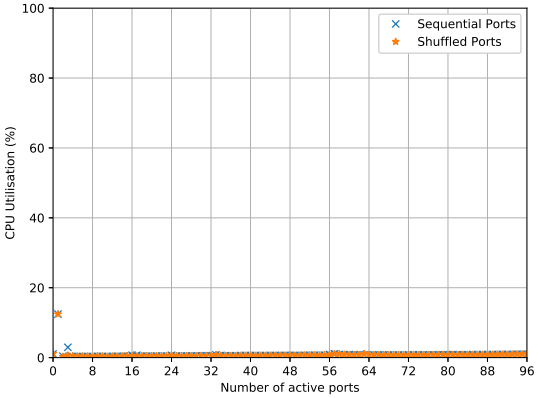


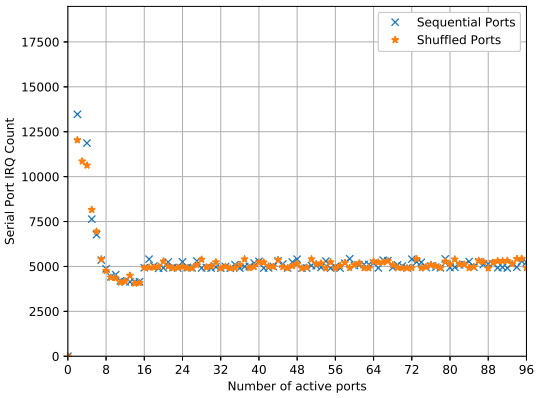




Sequential Ports
Shuffled Ports
Serial Port IRQ Count
Number of active ports

## 115 200 baud

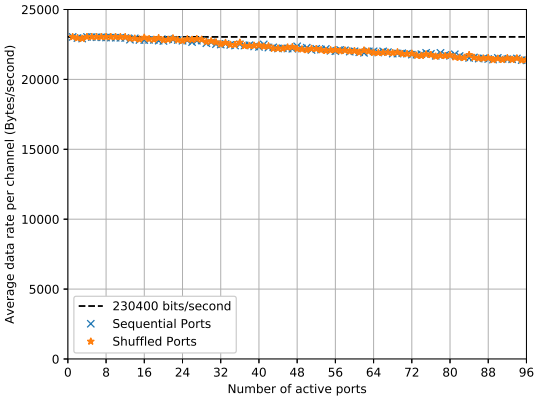

115200 bits/second
Sequential Ports
Shuffled Ports
Average data rate per channel (Bytes/second)
Number of active ports

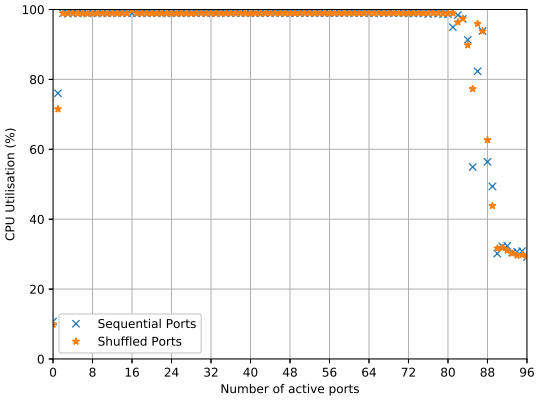

Sequential Ports
Shuffled Ports
CPU Utilisation (%)
Number of active ports


Sequential Ports
Shuffled Ports
Serial Port IRQ Count
Number of active ports

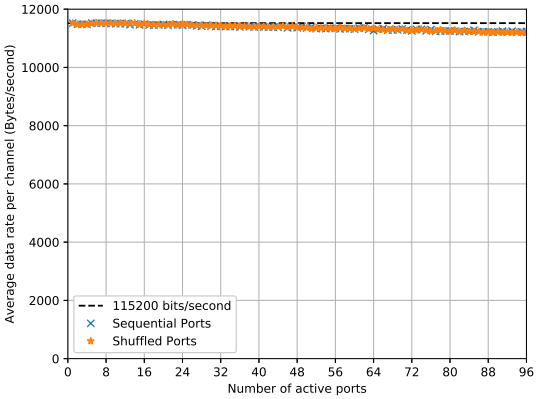
## B.4 Intel i7-960, Single Core

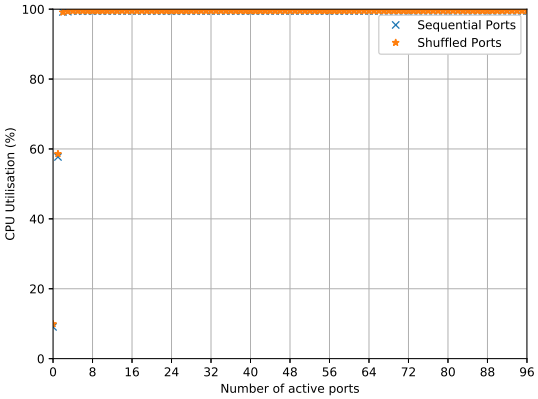
230 400 baud



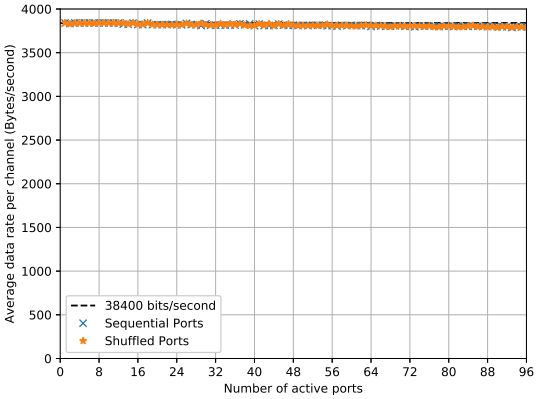



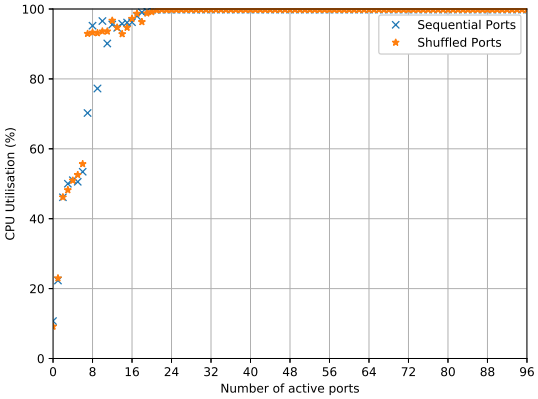

115 200 baud




38 400 baud

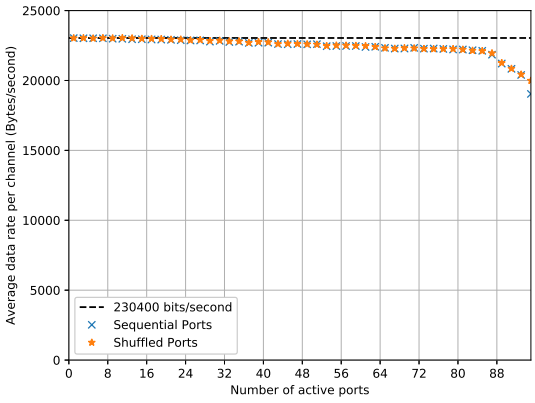


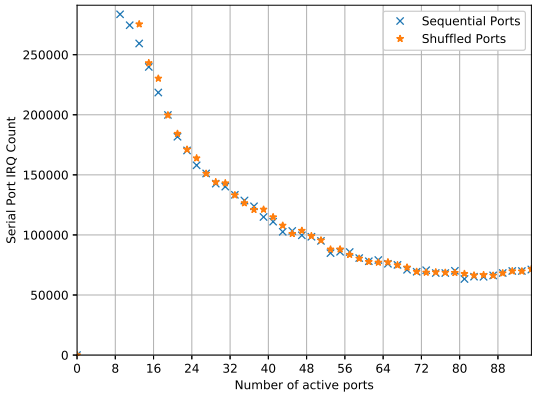



## B.5 Intel i7-960, Single Core, Artificial CPU Load

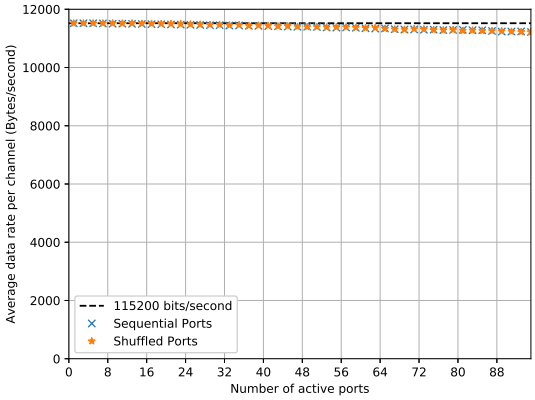
The conditions of this test force the CPU to always be at 100 % utilisation. The plots for this condition are not included.

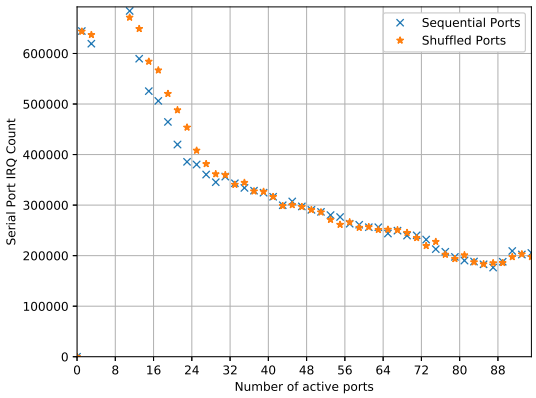
230 400 baud




115 200 baud



# Appendix C

# Monthly Reflections

The first three monthly reflections can be found overleaf. Specifically,



The remainder of this page has been intentionally left blank for formatting purposes.

## Key Learning Events

| **Learning Event** | EA Stage 1 Competencies | Description of learning event |
|---|---|---|
| Safety Inductions and On-boarding | 3.1 - *Ethical conduct and professional accountability.*<br>3.5 - *Orderly management of self, and professional conduct.* | As part of getting settled in at Opengear, I was given a tour of the office building and underwent a brief overview of the safety protocols for the lab space. As a result, I became quite familiar with the facilities and the equipment inside the electronics lab space. Although I don't expect to be in the lab space often, it's an incredibly valuable resource. |
| Briefing Meetings | 1.4 - *Discernment of knowledge development and research directions within the engineering discipline.*<br>2.4 - *Application of systematic approaches to the conduct and management of engineering projects.*<br>3.2 - *Effective oral and written communication in professional and lay domains.* | I spent much of my first and second days discussing the project with the people I'll be working most with. Other people, aside from my supervisor, at Opengear were aware of my project, yet there were many differing ideas on exactly what the project was. Naturally, after discussing the project and a direction to move in, this has been resolved. It's quite interesting to see the sort things which could be potentially included in the project if time permits. |
| Weekly Progress Presentations | 3.2 - *Effective oral and written communication in professional and lay domains.*<br>3.3 - *Creative, innovative, and proactive demeanour.* | See below. |
| Choice of development tools and environment | 1.3 - *In-depth understanding of specialist bodies of knowledge within the engineering discipline.*<br>2.1 - *Application of established engineering methods to complex engineering problem solving.*<br>3.1 - *Ethical conduct and professional accountability.* | Early on in the project, I had to select the FPGA development environment I intended to work with. This involved making a trade-off between ease of development, available product documentation, and cost. After consulting with supervisors, the Xilinx AC701 development board was chosen. |
| Project Planning | 2.1 - *Application of established engineering methods to complex engineering problem solving.*<br>2.2 - *Fluent application of engineering techniques, tools, and resources.* | Any engineering project requires careful thought and planning in order to be successfully carried out, and this is no exception. In conjunction with selecting the right equipment for the project, I spent plenty of time planning and designing my solution for the project. |

## SEAL Analysis for Event 3 *Weekly Progress Presentations*

**Situation** *What was the new experience or challenge you faced and what happened to you?*

Every Wednesday I meet with my on-site supervisor and a few other stakeholders for my project to discuss my progress. This involves me giving a brief presentation about the work I've done and where I intend to spend my time until the next meeting. I am a naturally confident speaker in situations like this, especially coming from a background in performing arts, and did not anticipate any major issues with nerves.

**Effect** *What impact did it have on you and what were the consequences of this impact?*

Unfortunately in the first meeting I was startled, having lost track of time while refining some diagrams to present. Additionally, I didn't gather the necessary equipment to present in the normal boardroom so the presentation was held huddled around my workstation. This clearly isn't ideal and so I now make a point to bring diagrams and other work to demonstrate printed out in meetings.

**Action** *What actions did you take to deal with the challenge you faced and why did you do those things?*

Taking the time after each meeting to plan exactly what to have ready for the next meeting is proving invaluable. With proper planning, I don't have to deal with the possibility of needing to rush to finish something to show right up until the last minute. As a result, the quality of my presentations has definitely improved.

**Learning** *What have you learnt from the experience and how will you apply this in the future?*

Since that first meeting and in the future, I aim to be better prepared for presentations. This leads to an overall better quality of presentation and makes it more worthwhile for myself and for the other attendees.

Additionally, in these meetings I gain invaluable insights and feedback from the on-site supervisors. Each meeting is a learning experience of its own.

**Engineering Australia Stage 1 Competencies Developed**

1.3 - *In-depth understanding of specialist bodies of knowledge within the engineering discipline.*

2.1 - *Application of established engineering methods to complex engineering problem solving.*

3.1 - *Ethical conduct and professional accountability.*

## Key Learning Events

| **Learning Event** | EA Stage 1 Competencies | Description of learning event |
|---|---|---|
| Read through advanced training materials for Xilinx FPGAs. | 1.3 - *In-depth understanding of specialist bodies of knowledge within the engineering discipline.*<br>2.2 - *Fluent application of engineering techniques, tools, and resources.* | To help optimise the FPGA portion of my design, I spent about two or three days reading through advanced training materials for using Xilinx FPGAs. The materials covered techniques for defining and constraining clocks, as well as techniques for working in digital designs with asynchronous clocks. |
| Read the relevant sections of the AXI Bus specification. | 1.3 - *In-depth understanding of specialist bodies of knowledge within the engineering discipline.*<br>2.1 - *Application of established engineering methods to complex engineering problem solving.*<br>2.3 - *Application of systematic engineering synthesis and design processes* | The two main components for the FPGA portion of my design – the PCI Express block and the UART block – are connected using the ARM AXI Bus. As such, I was required to read through parts of the specification and observe the behaviour of *known-good* AXI components so I could correctly implement it. |
| Learned the basics of *SystemVerilog* for AXI VIP | 1.3 - *In-depth understanding of specialist bodies of knowledge within the engineering discipline.*<br>2.1 - *Application of established engineering methods to complex engineering problem solving.*<br>2.3 - *Application of systematic engineering synthesis and design processes* | See below. |
| Learned about `/proc/stat` and `/proc/interrupts` | 1.2 - *Conceptual understanding of the mathematics, numerical analysis, statistics, and computer and information sciences which underpin the engineering discipline.*<br>2.1 - *Application of established engineering methods to complex engineering problem solving*<br>2.2 - *Fluent application of engineering techniques, tools, and resources* | A major part of my solution to the task at hand is benchmarking both my own design and the existing design. Measuring the system throughput is a simple matter, however measuring how *hard* the CPU has to *work* is less straightforward. The Linux kernel provides two virtual files, `/proc/stat` and `/proc/interrupts` which provide information on how much time the CPU spends performing certain tasks. This project has required that I learn what the contents of these files implies and how to process that information. |
| Wrote a PCB power budget | 1.1 - *Comprehensive, theory based understanding of the underpinning natural and physical sciences and the engineering fundamentals applicable to the engineering discipline.*<br>2.3 - *Application of systematic engineering synthesis and design processes.* | In order to verify that my printed circuit board (PCB) will work the first time, I developed a *power budget* to calculate exactly how much power the design is expected to use. This required learning about and understanding key parameters of the components I'm using in my design, such as *power dissipation capacitance.* |

## SEAL Analysis for Event 3 *Learned the basics of* SystemVerilog *for AXI VIP*

**Situation** *What was the new experience or challenge you faced and what happened to you?*

Interfacing portions of the FPGA component of my design with a PCI Express component requires that I implement the Advanced Extensible Interface (AXI) in my design. Naturally, being a non-trivial interface, there were a few issues with my design. When I attempted to communicate with the hardware I had designed using the PCI Express block and the AXI interface, I routinely found that the system would *hang*. Therefore, I needed to more thoroughly test my implementation.

There is a module available which helps to automatically verify that an implementation of AXI is correct, called the AXI VIP[1]. However, my design was implemented entirely in the hardware description language VHDL, whereas the VIP core only exists in SystemVerilog form.

**Effect** *What impact did it have on you and what were the consequences of this impact?*

The key impact of the VIP core only existing in SystemVerilog is that it meant I had to re-engineer my testbed to use SystemVerilog instead of VHDL. I welcome the chance to learn a new *programming* language[2] when there's a good reason to.

**Action** *What actions did you take to deal with the challenge you faced and why did you do those things?*

In order to effectively test my design using the AXI VIP module, I quickly learned the basics of, and re-engineered my testbed to SystemVerilog. I did this so that I could then make complete use of the AXI VIP to test and debug my design. As it turns out, the design issue I was having was caused because I misread a few key sentences of the AXI specification.

**Learning** *What have you learnt from the experience and how will you apply this in the future?*

I have now learned the absolute basics of SystemVerilog, and how to use it to test designs based around the AXI interface using the AXI VIP module. As a result of debugging my design, I have discovered a few more *gotchas* in implementing AXI.

**Engineering Australia Stage 1 Competencies Developed**

1.3 - *In-depth understanding of specialist bodies of knowledge within the engineering discipline.*

2.1 - *Application of established engineering methods to complex engineering problem solving.*

2.3 - *Application of systematic engineering synthesis and design processes*

[1] Verification Intellectual Property

[2] Strictly speaking, neither VHDL nor SystemVerilog are *programming languages*, but are actually *hardware description languages*. The difference being that rather than describing an ordered sequence of events, HDLs describe the layout of physical hardware.

## Key Learning Events

| **Learning Event** | EA Stage 1 Competencies | Description of learning event |
|---|---|---|
| Learned the basics of Linux Kernel development. | 1.2, 1.3, 1.4, and 2.2<br>See below. | See below. |
| Developed a series of trivial kernel modules. | 1.2 - *Conceptual understanding of the mathematics, numerical analysis, statistics, and computer and information sciences which underpin the engineering discipline.*<br>2.3 - *Application of systematic engineering synthesis and design processes.* | As part of becoming familiar with Kernel development, I wrote a few small *toy* modules to explore different features and interfaces within the kernel. Specifically, I worked through simple examples involving character devices, the TTY[1] subsystem, and the PCI Express / DMA subsystem. |
| Became familiar with Message Signal Interrupts (MSI). | 1.2 - *Conceptual understanding of the mathematics, numerical analysis, statistics, and computer and information sciences which underpin the engineering discipline.*<br>1.4 - *Discernment of knowledge development and research directions within the engineering discipline.*<br>2.2 - *Fluent application of engineering techniques, tools, and resources.* | Hardware devices can wake up the host CPU by sending *interrupt* signals. In PCI Express, these interrupts are delivered *in-band*, in line with all the other PCI Express signals, rather than requiring a dedicated connection. I spent quite a lot of time stumbling trying to have these interrupts work correctly with the Linux driver.[2] |
| Debugged an FPGA design using JTAG. | 2.1 - *Application of established engineering methods to complex engineering problem solving.*<br>2.2 - *Fluent application of engineering techniques, tools, and resources.*<br>2.3 - *Application of systematic engineering synthesis and design processes.* | While developing my Linux driver, I discovered that there were performance issues prevalent in the FPGA portion of the design. From an initial glance it appeared that it was not possible to send enough data over the PCI Express link in its current configuration to saturate all of the serial ports. Probing the internal interfaces using JTAG[3] helped me discover exactly where the bottleneck was. |
| Learned about AXI Stream and how I can use it in my design. | 2.1 - *Application of established engineering methods to complex engineering problem solving.*<br>2.4 - *Application of systematic approaches to the conduct and management of engineering projects.* | The interface between my portion of the FPGA design and the PCI Express DMA IP core is specified by the AXI Stream protocol. Initially I had planned to use the memory mapped alternative however upon further research the streaming option appeared to be far more appropriate for this application. As it turns out, many of the internal interfaces in my design were actually compatible with AXI Stream, entirely by accident. This discovery has meant that I can integrate IP cores from Xilinx to enhance my design. |

[1] Tele-typewriter

[2] The issue turned out to be that the FPGA device did not have bus-mastering enabled – informally it was trying to *speak* whilst in *speak-only-when-spoken-to* mode.

[3] Joint Test Action Group, a standardised protocol to test and verify digital circuits

## SEAL Analysis for Event 1 *Learned the basics of Linux Kernel development*

**Situation** *What was the new experience or challenge you faced and what happened to you?*

My project can be cleanly divided into three major sections: the printed circuit board design, the FPGA design, and the Linux kernel driver. Developing drivers for the Linux kernel was not taught during any of the courses I have taken at UQ, and as such, I had to learn it myself. I am a strong programmer, however writing code in a new ecosystem always makes for a steep learning curve. Thankfully, the engineers at Opengear have plenty of expertise in this area.

**Effect** *What impact did it have on you and what were the consequences of this impact?*

The impact of not having experience with Linux kernel driver development is that I had to learn how to develop drivers for the Linux kernel. Additionally, once I had learned the basics of kernel driver development, I uncovered hidden bugs in the FPGA design – ones which did not appear under previous test conditions.

**Action** *What actions did you take to deal with the challenge you faced and why did you do those things?*

I spent a number of days reading chapters from the book Linux Device Drivers[4], reading documentation distributed with the Linux kernel source code, and working through examples from both. I developed miniature kernel modules to help solidify and test what I had learned. By actively developing modules for the Linux kernel, I had learned how to do it.

**Learning** *What have you learnt from the experience and how will you apply this in the future?*

By doing this, I have now learned how to develop modules for the Linux kernel. Despite being a specialised skill, developing code to run inside the kernel is very similar to writing code for an embedded system – most of the typical support supplied to application-level code is entirely absent and the purpose of the code is usually to interface directly with external hardware. System-level development like this is a transferable skill for embedded systems which I can apply in the future.

**Engineering Australia Stage 1 Competencies Developed**

1.2 - *Conceptual understanding of the mathematics, numerical analysis, statistics, and computer and information sciences which underpin the engineering discipline.*

1.3 - *In-depth understanding of specialist bodies of knowledge within the engineering discipline.*

1.4 - *Discernment of knowledge development and research directions within the engineering discipline.*

2.2 - *Fluent application of engineering techniques, tools, and resources.*

[4] 2005, Linux Device Drivers, Third Edition, Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. Published by O'Reilly Media.

## Key Learning Events

| **Learning Event** | EA Stage 1 Competencies | Description of learning event |
|---|---|---|
| Implemented FPGA transfers using DMA. | 1.3 - *In-depth understanding of specialist bodies of knowledge within the engineering discipline.*<br>2.1 - *Application of established engineering methods to complex engineering problem solving.*<br>2.3 - *Application of systematic engineering synthesis and design processes* | Over the last month, I've made significant progress in the development of the Linux driver. To do this, I've had to implement transfers using direct memory access to the FPGA device. As a result, I've gained a lot of experience debugging PCI Express systems, and DMA. |
| Designed a very large PCB. | 1.2 - *Conceptual understanding of the mathematics, numerical analysis, statistics, and computer and information sciences which underpin the engineering discipline.*<br>2.1 - *Application of established engineering methods to complex engineering problem solving.*<br>2.3 - *Application of systematic engineering synthesis and design processes* | I finished designing the 96-port PCB in this last month. This board measures 8 inches by 14 inches, with six layers of routing, and over 200 components on-board. This is the first time I've designed a PCB even close to this scale, and has been a thoroughly valuable learning experience. In particular, experience coordinating with other people to acquire all the parts to build the board as been most valuable. |
| Designed and debugged analogue circuitry. | See below. | See below. |
| Significantly improved my VHDL testbed. | 2.1 - *Application of established engineering methods to complex engineering problem solving.*<br>2.3 - *Application of systematic engineering synthesis and design processes.*<br>2.4 - *Application of systematic approaches to the conduct and management of engineering projects.* | As any design becomes increasingly complex, more issues in its implementation are inevitably found. While developing the Linux driver, several *bugs* were identified in the FPGA design which did not appear under the contrived conditions of the original testbench. Obviously it is impractical to test every possible configuration of settings and as such testing to ensure a perfectly reliable system is incredibly difficult. While various components were being developed, they each had their own individual testbed, and these testbeds often failed to accurately emulate the rest of the system, and as such a grand unified testbed was created, to test the whole system at once. Every time I find something new which caused a problem in the design, it is a new learning experience. |

## SEAL Analysis for Event 3 *Designed and Debugged a Complex Analogue Circuit*

**Situation** *What was the new experience or challenge you faced and what happened to you?*

Two PCBs are being built as part of this project: the full 96-port board and a smaller, 8-port board with a high-speed analogue front-end for measuring current draw over time. The analogue front-end was designed to measure spikes of current entering the device of up to 40 mA occurring 18 million times per second. Unfortunately, this estimate was far too low, and as such the amplifiers in the circuit were not able to adequately measure the expected currents. Specifically, the first operational amplifier in the circuit did not have enough gain to output a meaningful output voltage.[1]

**Effect** *What impact did it have on you and what were the consequences of this impact?*

The result of the circuit not working is that I needed to find a way to correct it on the board, or find another way to measure the current draw.

**Action** *What actions did you take to deal with the challenge you faced and why did you do those things?*

The immediate cause for the circuit not functioning as expected was difficult to uncover. Based on some quick measurements, it was fairly clear that the second amplifier in the circuit was working as expected, however there was an issue with the first (the *gain* stage). Increasing the gain of the first amplifier would mean that the amplifier was no longer able to output a reliable signal – the circuit was already operating at the limit for the amplifier.

I took very low frequency measurements using a multimeter across the shunt resistor to take some readings for current draw. Although I had no way of viewing the actual peaks, the low frequency measurements were reassurance enough that the larger board would operate as expected.

**Learning** *What have you learnt from the experience and how will you apply this in the future?*

In the end, the gain was marginally increased and this still was not enough to *fix* the circuit. I see now that using a single ended amplifier topology for measuring such a small signal was unwise and won't make that mistake again in the future.

**Engineering Australia Stage 1 Competencies Developed**

1.2 - *Conceptual understanding of the mathematics, numerical analysis, statistics, and computer and information sciences which underpin the engineering discipline.*

1.3 - *In-depth understanding of specialist bodies of knowledge within the engineering discipline.*

2.1 - *Application of established engineering methods to complex engineering problem solving.*

2.3 - *Application of systematic engineering synthesis and design processes.*

2.4 - *Application of systematic approaches to the conduct and management of engineering projects.*

[1] The schematic for this circuit can be found in the Interim Report, Figure 5.10.

## Key Learning Events

| **Learning Event** | EA Stage 1 Competencies | Description of learning event |
|---|---|---|
| Profiled a kernel module | See below. | See below. |
| Debugged code by understanding x86 assembler output | 1.2 - *Conceptual understanding of the mathematics, numerical analysis, statistics, and computer and information sciences which underpin the engineering discipline.*<br>1.3 - *In-depth understanding of specialist bodies of knowledge within the engineering discipline.*<br>1.4 - *Discernment of knowledge development and research directions within the engineering discipline.* | Whenever my kernel driver crashes – a frequent occurrence during development – tracing information is printed to the screen to help isolate the source of the error. One particular piece of information given is the assembly-language representation *CPU instruction* currently being executed. To help me find issues faster, I took the time to briefly study some of the x86 instruction set. Reading about the way |
| Ported my driver from x86 to ARM | 1.1 - *Comprehensive theory based understanding of the underpinning natural and physical sciences and the engineering fundamentals applicable to the engineering discipline.*<br>1.3 - *In-depth understanding of specialist bodies of knowledge within the engineering discipline.*<br>1.5 - *Knowledge of engineering design practice and contextual factors impacting the engineering discipline.* | To create a true apples-for-apples comparison between my final solution and the original product, I needed to connect the hardware I had designed to an Opengear mainboard. This meant that the far more powerful CPU in my development machine was unable to skew the test results. Taking code written for one CPU and making it work on another is a tricky process. The process is made even more difficult because of the way Opengear locks down their products. |
| Final project presentation at the university | 3.2 - *Effective oral and written communication in professional and lay domains*<br>3.3 - *Created, innovative, and proactive demeanour.*<br>3.5 - *Orderly management of self, and professional conduct.* | One major piece of assessment for this course was the final presentation with supervisors and course staff. Unfortunately this presentation did not go entirely as well as it could have, largely because I misunderstood the requirements of the task and prepared a presentation which was far too technically focussed. Frankly, it was quite a wake-up call for ensuring that I completely understand the purpose of an assessment item before starting it. |

## SEAL Analysis for Event 1 *Profiled a kernel module*

**Situation** *What was the new experience or challenge you faced and what happened to you?*

System performance is a major consideration in this project. When the system was first *fully functional*, its performance well below that of the original it sought to replace. The situation is made even more serious because the *bottleneck* component in the design – the CPU – is much faster on my development machine than on the original console servers and performance was still abysmal.

**Effect** *What impact did it have on you and what were the consequences of this impact?*

The biggest impact was that this problem defined the last two weeks of the project, and forced me to reconsider the slower parts of the code.

The main consequence of the poor performance is that the resulting product did not meet performance specifications set out at the start of the project.

**Action** *What actions did you take to deal with the challenge you faced and why did you do those things?*

The first step in fixing performance issues is to identify exactly what the cause of the issue is. I was able to see that the CPU was almost 100% active while data was being sent, so I figured that it was likely that is where the issue is. As such, I researched tools for *profiling* my driver but discovered that I could get most of the functionality that I needed by building a simple software *stopwatch* to time all of the slow parts.

**Learning** *What have you learnt from the experience and how will you apply this in the future?*

Although I decided they were overkill for this particular application, I discovered a bunch of tools which can be used to help analyse my code and make it faster. I've now also gained much experience in improving the running speed of critical code and this will help me to write better code and design better systems in the future.

**Engineering Australia Stage 1 Competencies Developed**

1.2 - *Conceptual understanding of the mathematics, numerical analysis, statistics, and computer and information sciences which underpin the engineering discipline.*

1.3 - *In-depth understanding of specialist bodies of knowledge within the engineering discipline.*

1.4 - *Discernment of knowledge development and research directions within the engineering discipline.*

2.1 - *Application of established engineering methods to complex engineering problem solving.*

2.2 - *Fluent application of engienering techniques, tools, and resources.*

# Appendix D

# 96-Port PCB Power Budget

A power budget was constructed based on parameters from the data sheets for the components used in the design:

- 74LVC595A,
- 74LV165,
- SN74LVC125,
- TPS22917, and
- passive components where appropriate.

The power budget also assumes abnormally high load conditions, switching frequencies, and temperatures. All of this was done to ensure that any design based on the estimation provided is definitely sufficient for the real load.

The final result for power draw with 96 active ports is 1.393 W. By assuming temperatures well outside of the normal operational range, and assuming highly capacitive loads, an even more conservative figure is 5.208 W.

The power budget can be found overleaf.

# Power budget for 96 port UART board

June 26, 2018

## 1 Estimated Power Consumption for 96 port UART board

### 1.1 Initial assumptions

- 2.5V input voltage
- The INA219 has it's own dedicated 3.3V rail, with negligible leakage between 2.5V and 3.3V rails.

```
In [66]: import numpy as np
         import matplotlib.pyplot as plt
         from scipy import stats

         VCC = 2.5 # Volts
         FMAX = 73728000 # Hz
         NUARTS = 96
```

### 1.2 74LV165A -- Input SR

```
In [67]: def LV165(VCC, FMAX):
             # STATIC POWER DISSIPATION PER 74LVC165A
             ICC = np.asarray([0.05, 5]) * 1e-6; # page 7
             LV165_P_Static = VCC * ICC


             # DYNAMIC POWER DISSIPATION PER 74LVC165A
             #      -- from datasheet, page 10
             Cpd = 24e-12
             fi = FMAX
             fo = np.asarray([230400*8, fi])
             Noutputs = 2
             CL = 1e-12 * np.ones((1,Noutputs))

             LV165_P_Dynamic = Cpd * VCC**2 * fi + \
                 np.sum(np.outer(CL, fo) * VCC**2, axis=0)

             # TOTAL POWER DISSIPATION
             LV165_P_Total = LV165_P_Static + LV165_P_Dynamic
```

```
            return LV165_P_Total

        print("Total power dissipation per 74LV165A: " +
              "{:.03f}-{:.03f} mW".format(*1000*LV165(VCC,FMAX)))
        print("Total power dissipation for {} UARTS: ".format(NUARTS) +
              "{:.03f}-{:.03f} W".format(*(LV165(VCC,FMAX) * NUARTS/2)))
```

```
Total power dissipation per 74LV165A: 11.082-11.993 mW
Total power dissipation for 96 UARTS: 0.532-0.576 W
```

### 1.3 74LVC595A -- Output SR

```
In [68]: def LVC595A(VCC, FMAX):
             # STATIC POWER DISSIPATION PER CHIP
             ICC = np.asarray([0.1, 10]) * 1e-6; # page 7
             DICC = np.asarray([5, 500]) * 1e-6; # page 7
             LVC595A_P_Static = (ICC + (5*DICC)) * VCC;

             # DYNAMIC POWER DISSIPATION PER 74LVC595A
             Cpd = 45e-12
             fi = 73.728e6 / 2 / 2
             fo = np.asarray([230400, fi])
             CL = 80e-12 * np.ones((1, 8))
             NSwitching = 4
             LVC595A_P_Dynamic = (Cpd * VCC**2 * fi * NSwitching)\
                 + np.sum(np.outer(CL, fo) * VCC**2,axis=0)

             # TOTAL POWER DISSIPATION
             LVC595A_P_Total = LVC595A_P_Static + LVC595A_P_Dynamic

             return LVC595A_P_Total

         print("Total power dissipation per 74LVC595A: {:.03f}-{:.03f} W"\
               .format(*LVC595A(VCC,FMAX)))
         print("Total power dissipation for {} UARTS: {:.03f}-{:.03f} W"\
               .format(NUARTS, *(LVC595A(VCC,FMAX) * NUARTS/2)))
```

```
Total power dissipation per 74LVC595A: 0.022-0.101 W
Total power dissipation for 96 UARTS: 1.043-4.835 W
```

### 1.4 Pull-up resistors on RX signal lines

```
In [69]: Npullup = NUARTS * 4
         Rpullup = np.asarray([1.01, 0.99])*10e3 # Ohms
```

```
P_Pullup = VCC**2 / Rpullup

print("Power wasted in pullup resistors: {:.03f}-{:.03f} W"\
      .format(* (1e3*P_Pullup)))
```

```
Power wasted in pullup resistors: 0.619-0.631 W
```

### 1.5 Control Signal Buffers

```
In [70]: Nbuffers = NUARTS/6

         def SN74LVC125A(VCC, FMAX):
             Cpd = 11.3e-12
             pd = Cpd * VCC**2 * FMAX + 4 * (VCC**2 * 5e-12)

             return np.asarray([pd, pd])

         print("Total power dissipation per 74LVC595A: {:.03f}-{:.03f} mW"\
               .format(*1e3*SN74LVC125A(VCC,FMAX)))
         print("Total power dissipation for {} UARTS: {:.03f}-{:.03f} mW"\
               .format(NUARTS, *(1e3*SN74LVC125A(VCC,FMAX)*Nbuffers)))
```

```
Total power dissipation per 74LVC595A: 5.207-5.207 mW
Total power dissipation for 96 UARTS: 83.313-83.313 mW
```

### 1.6 Total estimated power dissipation

```
In [71]: def TotalPower(VCC, FMAX, NUARTS):
             Ptot = ((LV165(VCC, FMAX/2) + LVC595A(VCC, FMAX)) * NUARTS/2)\
                 + P_Pullup + (SN74LVC125A(VCC, FMAX)*NUARTS/6)
             Itot = Ptot / VCC;
             return (Ptot, Itot)

         Ptot, Itot = TotalPower(VCC, FMAX, NUARTS)

         TPS_Ron = np.asarray([ 80e-3, 125e-3]) # 125 mOhms @ 1.8V, 100 deg Celcius
         R_shunt = 100e-3

         Vdrop = (R_shunt + TPS_Ron) * Itot

         print("Total power = {:.03f} - {:.03f} W".format(*Ptot))
         print("Total current @ {:.03f} V is {:.03f}-{:.03f} mA"\
               .format(VCC, *1e3*Itot))
         print("Current per port is {:.03f} - {:.03f} mA"\
               .format(*1e3*Itot/NUARTS))
         print("Voltage drop in supply circuit: {:.03f} to {:.03f} mV"\
               .format(*Vdrop))
```

```
Total power = 1.393 - 5.208 W
Total current @ 2.500 V is 557.216-2083.022 mA
Current per port is 5.804 - 21.698 mA
Voltage drop in supply circuit: 0.100 to 0.469 mV
```

```
In [72]: x = np.linspace(0,96,20)
         y1 = np.asarray([TotalPower(VCC, FMAX, x)[0] for x in x])
         y2 = np.asarray([TotalPower(VCC, FMAX, x)[1] for x in x])

         fig, ax = plt.subplots(2,1,sharex=True)
         ax[0].plot(x, y1)
         ax[0].set_xticks(np.arange(0, 97, 16))
         ax[0].set_ylabel('Power\nConsumption (W)')
         ax[0].legend(['Typical', 'Max'])
         ax[0].grid(True)

         ax[1].plot(x, y2)
         ax[1].set_xticks(np.arange(0, 97, 16))
         ax[1].set_ylabel('Current Draw\nat 2.5V (A)')
         ax[1].set_xlabel('Fitted ports')
         ax[1].legend(['Typical', 'Max'])
         ax[1].grid(True)

         _ = ax[0].set_xlim(0,97)
```