A Complete Open Source Network Stack For BlackParrot

Yuan-Mao Chueh

A thesis
submitted in partial fulfillment of the
requirements for the degree of

Master of Science in Electrical Engineering

University of Washington
2022

Committee:
Michael Taylor
Scott Hauck

Program Authorized to Offer Degree:
Electrical and Computer Engineering

University of Washington

Abstract

A Complete Open Source Network Stack For BlackParrot

Yuan-Mao Chueh

Chair of the Supervisory Committee:
Michael Taylor
Department of Computer Science and Engineering

Dennard scaling has come to an end. General-purpose architecture now can hardly have major improvements in power efficiency. Therefore, recently researchers have been actively coming up with hardware designs that do only a limited number of tasks but with great efficiency. Those designs are called hardware accelerators. The future hardware will equip more and more such accelerators and the general-purpose cores will become the hosts of those accelerators. BlackParrot is a Linux-Capable RISC-V Multicore that strives to be the hardware accelerator host widely used by the world. Although BlackParrot has competitive performance and good power efficiency as compared to other RISC-V Cores, the lack of open source I/O devices significantly limits its applications, for example, supporting standard Linux distributions. In my thesis work, a complete network solution has been brought to BlackParrot. This includes creating an Ethernet controller, which contains a ported open source Ethernet MAC, testing a RISC-V PLIC module, porting two embedded libraries that together allow making secure SSL connections, enabling the use of the off-the-shelf network stack from Linux kernel 5.15, and also supporting a Linux distribution from Yocto by doing NFS mounting. Both the software and the hardware are completely open source, and both of them have been validated on FPGA.

Contents

List of Figures

List of Tables

1 Introduction

Due to the end of the Dennard scaling[2], the power density of silicon chips has no longer been constant and hence forces the research focus to switch from general-purpose architecture, which can process on various kinds of workload, to domain-specific architecture, which does only a small subset of tasks with orders of magnitude improvement in both performance and cost over the general-purpose one. This unavoidable movement has brought plenty of hardware accelerators to the world, including the famous google TPU[3] and Cerebras WSE-2[4].

BlackParrot[5] is a BSD-licensed, fully open-source RISC-V multicore that is designed to be the default host processor for hardware accelerator chips. It has top performance among open-source cores in its efficiency class. However, despite the fact that BlackParrot is capable of running Linux kernel, it cannot run standard Linux distributions due to the lack of open source I/O devices. In order to make BlackParrot a fully functional SoC and support more software, the goal of my thesis work was to equip BlackParrot with Ethernet Connectivity. By doing this, BlackParrot would be able to connect to the Internet and run a standard Linux distribution by doing NFS mounting.

This thesis summarizes my works that try to address the above issue. For the hardware side, I have done the following three things: First, the RISC-V PLIC module from OpenTitan[6] has been tested with BlackParrot. Second, the 1G Ethernet MAC module, eth_mac_1g_rgmii_fifo, from Alex Forencich[7] has been ported to BlackParrot. Third, the S-mode external interrupt line has been added to BlackParrot. Extra efforts have been made to ensure all the hardware designs are largely technology-independent, which means developers only need to replace the small, technology-dependent parts of the designs in order to port these designs to different technology nodes or different FPGA boards.

For the software side, I have ported the TCP/IP library lwIP[8] and the SSL library wolfSSL[9] to BlackParrot. These lightweight libraries provide all the necessary functionalities to make secure network connections in embedded applications. I have also created three examples for the demonstration of the libraries. The first example is a collection of

TCP/SSL server/client programs, while the second and third examples are the boot loaders for the BlackParrot System on Module. Besides those libraries, I have also upgraded both Linux and OpenSBI in the BlackParrot SDK[10] to a higher version (from Linux 5.8 and OpenSBI 0.7, to Linux 5.15 and OpenSBI 1.0). The Liteeth driver[11] and the PLIC interrupt controller driver[12] from Linux kernel 5.15 have also been tested on the combination of BlackParrot, the Ethernet controller and the PLIC module. Both the Linux drivers and the Linux kernel can now work out-of-box on BlackParrot, which means no custom changes are needed. Finally, I was able to NFS boot a default Linux distribution from Yocto[13] on BlackParrot. Both the hardware and software components are fully free and open-source, and they have all been validated on an FPGA.

2 Background

2.1 OpenSBI

OpenSBI[14] is an open-source project that provides reference implementation of RISC-V SBI specifications[15], which provides a standard interface between a M-mode firmware and an S-mode operating system (for example, Linux). OpenSBI handles various kinds of things including instruction emulation, console operations and misaligned memory access. In our research group, an SBI implementation for BlackParrot was made based on the provided template implementation from OpenSBI. Our final SBI binary directly includes the entire Linux kernel binary as its payload and the SBI binary serves as the first bootloader running in M-mode.

2.2 BlackParrot

BlackParrot (BP) is an open-source Linux-capable multicore that supports RV64G and three privilege levels – machine, supervisor and user mode. It is licensed under BSD-3. It has an 8-stage, in-order pipeline and shows competitive performance among other RISC-V processors. It has a modular architecture that consists of three parts: front end, back end and memory end. This design has allowed contributors to work independently without breaking the others. For debugging, the tool Dromajo[16] has been ported to BlackParrot and it allows both fast software simulation and hardware co-simulation, which is handy for catching bugs at both software and hardware levels.

2.3 ZynqParrot

ZynqParrot[17] is a framework that can be built in every Zynq SoC[18] that contains an ARM processing system(PS) (for running Linux and other applications) and Xilinx programmable logic(PL) (for running synthesized design). There are several AXI connections between PS and PL. This enables the user to run a program in PS and get the outputs from PL. There is also an AXI connection between PL and the DRAM module in PS. This allows the digital design to access the memory in PS from PL. The above two features of the Zynq SoC are the key enablers of ZynqParrot.

Under the ZynqParrot framework, BlackParrot runs in PL. Whenever BlackParrot accesses its I/O interface, it will send read/write requests from PL to PS through an AXI connection. A program running on the PS will then either display the read data to the user or writes the data entered by the user back to the PL, depending on the decoding results of those requests. BlackParrot also leverages the DRAM module in PS through an AXI connection. The DRAM module serves as the backend of the Blackparrot L2 cache.

Besides the above infrastructure, the ZynqParrot framework also provides an environment for fast simulation of running a program in BlackParrot. This feature is helpful when developing hardware peripherals on ZynqParrot.

2.4 Litex

Litex[19] is a framework that enables efficient and convenient build of FPGA SoCs. The framework has supported several FPGA boards and provides many common components seen in an FPGA SoC, such as various CPUs, AXI bus, LiteDRAM, LitePCIe, LiteSATA and Liteeth. Although the Litex framework uses Migen as its Hardware Description Language, which is not as common as Verilog/SystemVerilog, the framework has provided a mechanism to translate Migen to Verilog. This makes the framework work seamlessly with the traditional development flow.

3 Hardware

3.1 Ethernet Controller

3.1.1 Introduction

According to the OSI model (figure 1)[20], there are 7 abstraction layers in computer networking, from the physical layer (layer 1) to the application layer (layer 7). The well-known HTTP protocol is at the application layer, while the TCP, IP protocols are at the transport layer and the network layer, respectively. The Ethernet protocol defined by IEEE 802.3 involves both the physical layer and the data link layer. The Ethernet physical transceiver (PHY) operates at the physical layer. It sends and receives the analog signals on the wired cable, and with different types of physical media (for example, copper in 1Gbps, optical fiber in 10Gbps speed case), different types of PHYs are used. The Ethernet MAC operates at the data link layer. It sends/receives the raw Ethernet frames to/from the PHY.

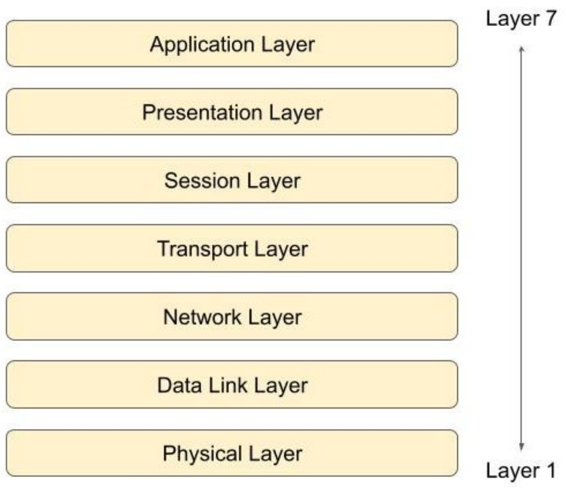


Figure 1: The OSI Model

Figure 2 shows the overview of the typical Ethernet hardware. The PHY device is a dedicated external module that is not within the pure digital design boundary. The original standard interface between MAC and PHY is called the Media-Independent Interface (MII). It allows the MAC module to connect to different types of media, and hence different types of PHYs, without the need to redesign the MAC hardware. Besides MII, there are other variants as well, including GMII/RGMII, which are commonly seen in 1G Ethernet, and XGMII, which appears in 10G Ethernet.

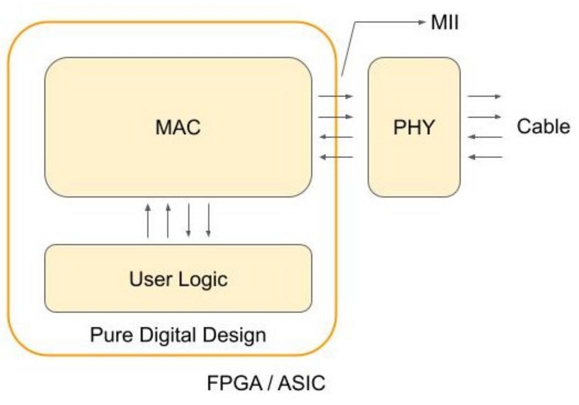


Figure 2: Ethernet Hardware Overview

The MAC module receives the layer 2 Ethernet frame (shown in figure 3) from the user logic. It then adds the preamble, start frame delimiter (SFD) and frame check sequence (FCS) to the frame before sending it to PHY. Frame check sequence is used to detect errors in the Ethernet frame. If there is an error, the receiver will just drop the Ethernet frame.

The user logic can be anything from simple Ethernet control logic to a complex TCP offload engine, depending on the specific need. The Ethernet control logic could include a hardware module that generates interrupts, a DMA engine that transfers the packets, and some control/status registers that provide the MAC access to the users. Together the Ethernet control logic and the MAC module form an Ethernet controller.

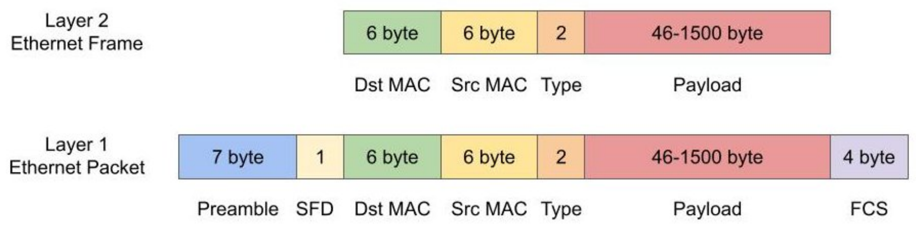


Figure 3: Ethernet Frame

3.1.2 Goals

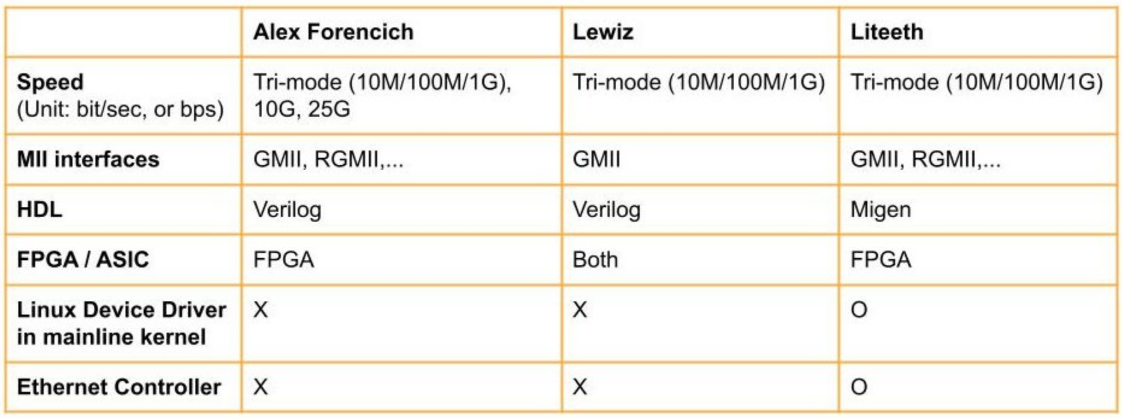
The Ethernet controller is designed with the four following goals: 1. The work should be fully open-source and leverage as many existing high quality, off-the-shelf projects as possible in order to minimize the hardware development cost as much as possible. 2. The work should be fully compatible with the existing Linux Ethernet device driver so that the entire software stack would work seamlessly with my work without the need of any custom changes. 3. The work is intended for use in both FPGA and ASIC, i.e., it should be synthesized in both ASIC and FPGA with only minimal technology-dependent primitives for I/O. 4. The work should support both 100Mbps and 1Gbps as they are the most common speeds seen in daily life.

3.1.3 Hardware Selection

Ethernet MAC Selection

In this thesis work, several MAC modules have been considered: Liteeth[21], LeWiz[22] and Alex[7]. As pointed out in figure 4, all the three MAC cores can achieve 100M and 1G Ethernet since they all support tri-mode, which includes 10M, 100M and 1G. However, only Liteeth comes with mainline Linux (starting from Linux 5.15) support. Although Liteeth could offer such a big benefit, the cost of learning the Litex framework and new HDL language might offset the benefit. Since Liteeth is designed for FPGA and is written in a language that is not as popular as Verilog, changing the code to make it work for both ASIC and FPGA would likely require developers to learn a new language and be familiar with the Litex framework. Although the framework can convert Migen into Verilog, the auto-generated code is not human-readable and hence is not good for porting and development. Given that we are only interested in Liteeth alone, from both the development and maintenance perspectives, the overhead of learning a new language and platform for just a single module from Litex would go beyond its own advantages.

Another option is to leverage the Linux device driver from Liteeth and use the MAC module from either Alex or Lewiz, which are all written in (System)Verilog. Although Lewiz is claimed to support both FPGA and ASIC, from the history of the repository on github and their responses to my previous filed issues, the project seems to have been inactive for some time and only few projects have adopted it. On the other hand, Alex's MAC module has been actively maintained so far and has been widely used. This leads



	Alex Forencich	Lewiz	Liteeth
Speed (Unit: bit/sec, or bps)	Tri-mode (10M/100M/1G), 10G, 25G	Tri-mode (10M/100M/1G)	Tri-mode (10M/100M/1G)
MII interfaces	GMII, RGMII,...	GMII	GMII, RGMII,...
HDL	Verilog	Verilog	Migen
FPGA / ASIC	FPGA	Both	FPGA
Linux Device Driver in mainline kernel	X	X	O
Ethernet Controller	X	X	O

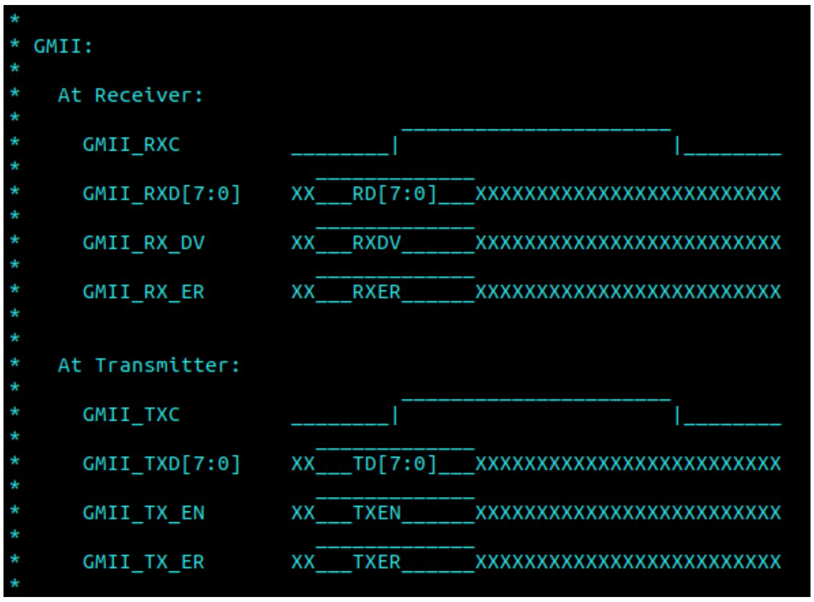
Figure 4: Comparison between Different MAC Modules

to the decision that the Liteeth Linux device driver in mainline Linux should be used for the software; the MAC module should be used to create a MAC module suitable for both ASIC and FPGA; and I should create our own Ethernet controller.

As for the MII interface, figure 5 and figure 6 show the GMII and RGMII interface respectively. Both GMII and RGMII are source-synchronous, which means the data signals are sent along with a clock signal generated from a transmitting node, rather than a global clock signal. In GMII, there are GMII_TXD and GMII_RXD for data transmission, both of which are 8 bit wide; there are also control signals for RX_DV (data valid), RX_ER (error) and TX_EN (enable), TX_ER (error). Whereas in RGMII, the data transmission lines for both RX and TX are cut down from 8 to 4 bits, which is achieved by transmitting the data on both the rising and falling edge of the clock signal, i.e., the first 4 bits on the rising edge and the last 4 bits on the falling edge. Besides, the number of control signals are also reduced by half: The RX_CTL/TX_CTL sends RX_DV/TX_EN on the rising edge and the 'exclusive or' of RX_DV/TX_EN and RX_ER/TX_ER on the falling edge. Since the RGMII interface significantly reduces the pin counts, it is the chosen MII interface in our implementation.

FPGA Board and PHY Module Selection

In order to run 1G Ethernet with RGMII interface on an FPGA, the board Zedboard[23], and the PHY module Avnet Network FMC Module[24] are selected. Zedboard has an



```
*
* GMII:
*
*   At Receiver:
*                               ______________________
*     GMII_RXC          ________|                      |________
*                         _____________
*     GMII_RXD[7:0]     XX___RD[7:0]___XXXXXXXXXXXXXXXXXXXXXXXXX
*                         _____________
*     GMII_RX_DV        XX___RXDV______XXXXXXXXXXXXXXXXXXXXXXXXX
*                         _____________
*     GMII_RX_ER        XX___RXER______XXXXXXXXXXXXXXXXXXXXXXXXX
*
*
*   At Transmitter:
*                               ______________________
*     GMII_TXC          ________|                      |________
*                         _____________
*     GMII_TXD[7:0]     XX___TD[7:0]___XXXXXXXXXXXXXXXXXXXXXXXXX
*                         _____________
*     GMII_TX_EN        XX___TXEN______XXXXXXXXXXXXXXXXXXXXXXXXX
*                         _____________
*     GMII_TX_ER        XX___TXER______XXXXXXXXXXXXXXXXXXXXXXXXX
*
```

Figure 5: GMII Signals

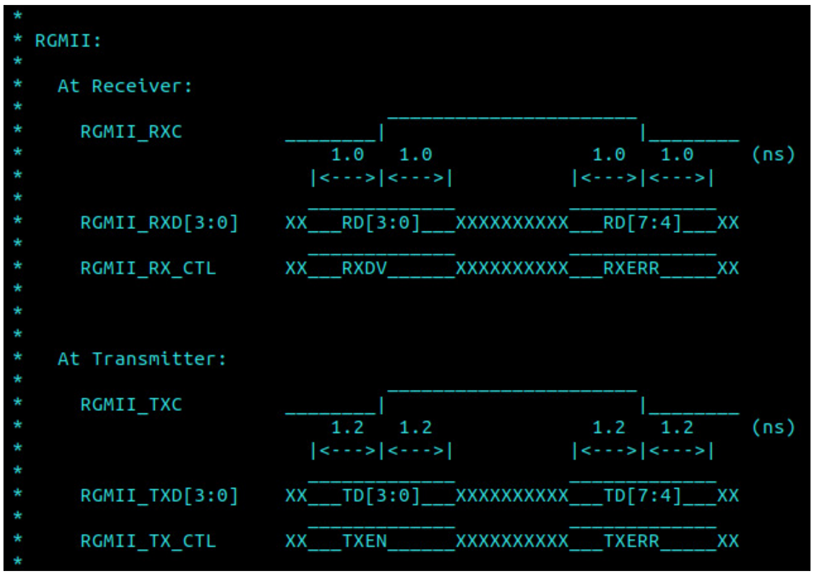
FMC LPC interface which can be connected to the network FMC module. With some proper pin assignment on Zedboard, the RGMII signals on the FPGA can be connected to the FMC interface, and then routed to the PHY module. The PHY module will then send the signals out to an Ethernet cable.

3.1.4 BlackParrot Ethernet Controller

Overview

The overview of the ethernet_controller module is shown in figure 7. All the modules that are used in both ASIC and FPGA are placed in this module. For those platform-dependent primitives, they are placed outside the ethernet_controller module and bundled together under the wrapper module, ethernet_controller_wrapper (see figure 8). For example, in the case of Zedboard, IDELAYCTRL and IDELAYE2 primitives are needed to meet the timing constraints for RGMII, therefore they are put under the wrapper module.

The eth_mac_1g_rgmii_fifo module is the Ethernet MAC module from Alex. The original version of the MAC module supports tri-mode and uses two 125 MHZ clocks (one with 90 degree shift) and some FPGA primitives (IDDR and ODDR) to deal with the RGMII signals. The original version also has a helpful status interface that users can



```
*
* RGMII:
*
*   At Receiver:
*
*                                ______________________
*     RGMII_RXC         ________|                      |________
*                           1.0   1.0              1.0   1.0     (ns)
*                         |<--->|<--->|          |<--->|<--->|
*                         _____________          _____________
*     RGMII_RXD[3:0]    XX___RD[3:0]___XXXXXXXXXX___RD[7:4]___XX
*                         _____________          _____________
*     RGMII_RX_CTL      XX___RXDV______XXXXXXXXXX___RXERR_____XX
*
*
*
*   At Transmitter:
*
*                                ______________________
*     RGMII_TXC         ________|                      |________
*                           1.2   1.2              1.2   1.2     (ns)
*                         |<--->|<--->|          |<--->|<--->|
*                         _____________          _____________
*     RGMII_TXD[3:0]    XX___TD[3:0]___XXXXXXXXXX___TD[7:4]___XX
*                         _____________          _____________
*     RGMII_TX_CTL      XX___TXEN______XXXXXXXXXX___TXERR_____XX
*
```

Figure 6: RGMII Signals

leverage to detect packet corruption at the Ethernet packet level. My main modifications to the MAC module are as follows: 1. Replace the FPGA primitives with the IDDR and ODDR modules from Basejump STL[25]. 2. Replace the two 125 MHZ clocks with one single 250 MHZ clock plus some generated clock logic. 3. Add IDELAYE2 primitives and use IOB packing to meet the timing constraints on Zedboard.

The MAC module supports tri-mode by detecting the speed of the external PHY, which can be 10Mbps, 100Mbp or 1Gbps, and runs at the exact same speed. If the speed of the external PHY changes during runtime, the speed of the Ethernet controller will also be adapted accordingly. For example, if the speed of PHY is changed from 100Mbps to 1Gbps, the Ethernet controller will detect the speed change (by looking at the clock cycles of the RGMII RX clock from PHY) and update its speed from 100Mbps to 1Gbps.

"ethernet_memory_map"

This module presents the memory map of the Ethernet controller (figure 9). The memory map is based on the memory map from the existing Linux Liteeth device driver.

For the transmission path: First, “TX Ready” bit is checked. If it is set, the TX buffer has space and can accept new packets. Second, write the packet to the “TX buffer” and the size of the packet to the “Length of the Transmitting Packet” register. Third, do a

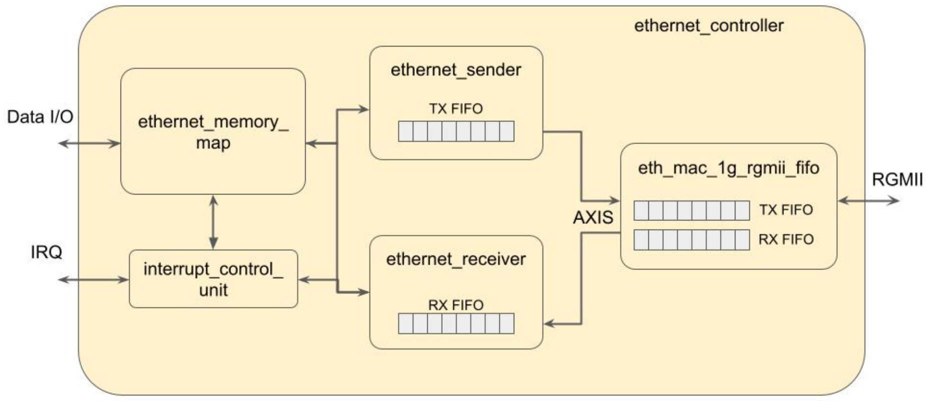


Figure 7: The ethernet_controller Module

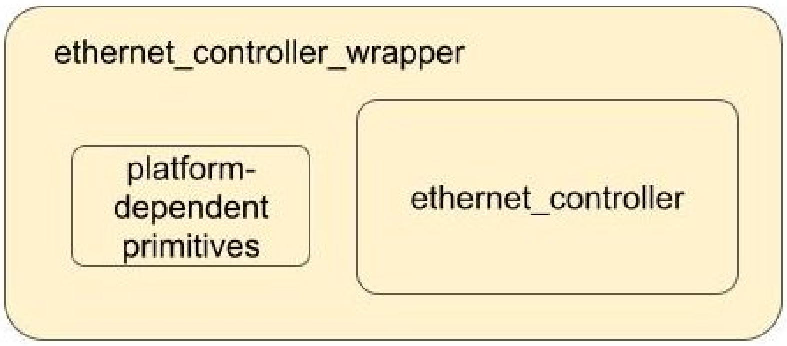


Figure 8: The ethernet_controller_wrapper Module

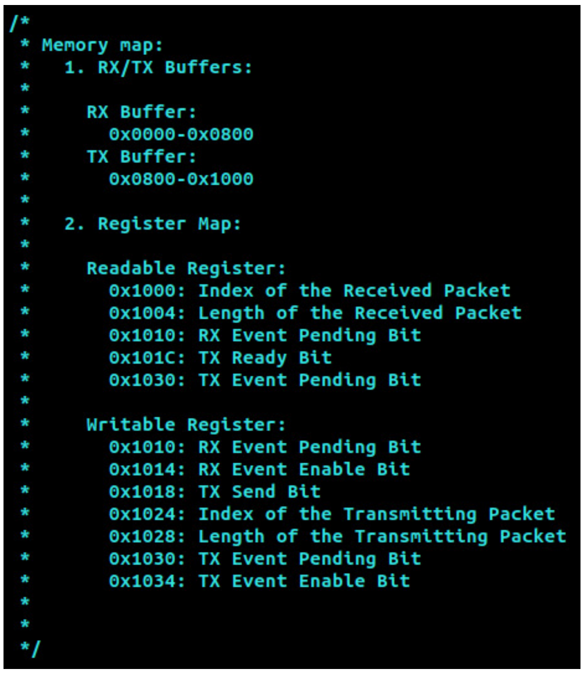
write (can be any value) to the "TX Send" bit to send the packet.

For the receive path: First, "RX Event Pending" bit is checked. If it is set, the RX buffer has packets to receive. Second, read the packet from the "RX buffer" and the size from the "Length of the Received Packet" register. Third, write "1" to the "RX Event Pending" bit to acknowledge the received packet.

The "TX Event Pending" bit is set when the status of the TX buffer goes from full to empty. This is used by the Liteeth Linux driver to notify Linux to resume the Network traffic. Writing "1" to it clears the bit.

When the enable bit is set, the corresponding interrupt is enabled. Otherwise it is disabled.

The "Index of the Received Packet" is unused and is always zero. The "Index of the transmitting packet" is also unused and all the values written to it are ignored. These fields are used as the FIFO pointers to the current RX/TX slots in the Liteeth Linux driver, but since in our design the main RX/TX buffers are in eth_mac_1g_rgmii_fifo and does not require the driver to maintain where the current slots are, these fields are deliberately tied to zero to hide those details from the software.



```
/*
 * Memory map:
 *   1. RX/TX Buffers:
 *
 *     RX Buffer:
 *       0x0000-0x0800
 *     TX Buffer:
 *       0x0800-0x1000
 *
 *   2. Register Map:
 *
 *     Readable Register:
 *       0x1000: Index of the Received Packet
 *       0x1004: Length of the Received Packet
 *       0x1010: RX Event Pending Bit
 *       0x101C: TX Ready Bit
 *       0x1030: TX Event Pending Bit
 *
 *     Writable Register:
 *       0x1010: RX Event Pending Bit
 *       0x1014: RX Event Enable Bit
 *       0x1018: TX Send Bit
 *       0x1024: Index of the Transmitting Packet
 *       0x1028: Length of the Transmitting Packet
 *       0x1030: TX Event Pending Bit
 *       0x1034: TX Event Enable Bit
 *
 *
 */
```

Figure 9: The Memory Map of the BlackParrot Ethernet Controller

"ethernet_sender" and "ethernet_receiver"

Since the TX and RX buffers in Liteeth Linux driver are presented as a random-access memory buffer, these modules are used to do the necessary conversion from the AXI stream interfaces to memory arrays. Both ethernet_sender and ethernet_receiver have a FIFO with 2 slots. Each slot can contain only 1 Ethernet packet, regardless of the size of the packet. The reason to have FIFOs with 2 slots is to allow both the AXI stream side and the ethernet_memory_map side of the hardware to operate at the same time.

"interrupt_control_unit"

This module generates two level-triggered interrupts, one for RX and the other for TX. The RX interrupt is asserted high when the RX buffer is not empty, while the TX interrupt is asserted high when the status of the TX buffer changes from full to non-full.

"eth_mac_1g_rgmii_fifo"

This is the core MAC module ported from Alex (figure 10). The Ethernet packets are sent and received through the AXI stream interface on the left, which connects to the inner part of the Ethernet controller. Right next to the AXI stream interface are the TX and RX FIFOs. They store the packets and do the clock domain crossing between the user logic clock and the Ethernet TX/RX clock. Inside the eth_mac_1g_rgmii module, eth_mac_1g sends/receives Layer 1 Ethernet frames to/from rgmii_phy_if through the GMII interface, while rgmii_phy_if does the conversion between GMII signals and RGMII signals. The RGMII signals then connect to the external PHY module.

rgmii_phy_if is responsible for maintaining the necessary phase difference between the RGMII_TXD/RGMII_TX_CTL and RGMII_TXC. As shown in the RGMII timing diagram in figure 6, instead of sending together, RGMII_TXC is center-aligned to every beat of RGMII_TXD and RGMII_TX_CTL, that is, the rising edge of RGMII_TXC is aligned with the first 4 bits of data plus TXEN and the falling edge is aligned with the last 4 bits of data plus TXERR. In order to meet such a requirement, from figure 11, rgmii_phy_if generates GMII_TXC (which generates RGMII_TXD, RGMII_TX_CTL), whose rising edges align with the rising edges of clk250_i, and generates RGMII_TXC, whose rising edges align with the falling edges of clk250_i. rgmii_phy_if also considers the latency between GMII_TXD and RGMII_TXD, so that the first RGMII data will always be sent along with a rising edge of RGMII_TXC. This can be seen in the same figure: 'd1d0' is an 8-bit GMII data and it is transmitted as two 4-bit RGMII data: 'd0', and then 'd1'. 'd0' is always followed by a rising edge of RGMII_TXC.

Figure 12 shows the rgmii_phy_if module. The speed_i signal in rgmii_phy_if comes from the GMII_RXC speed detection logic in eth_mac_1g_rgmii. It distinguishes between 2.5 MHZ, 25 MHZ and 125 MHZ and makes the speed of RGMII_TXC equal to GMII_RXC.

The original version of rgmii_phy_if uses IDDR and ODDR FPGA primitives and the 90-degree shifted clock to receive and send DDR signals. In order to make the design suitable for both FPGA and ASIC, those primitives and the clock signal have been replaced with the clk250_i signal (and its generated clock signals) and synthesized versions of IDDR and ODDR.

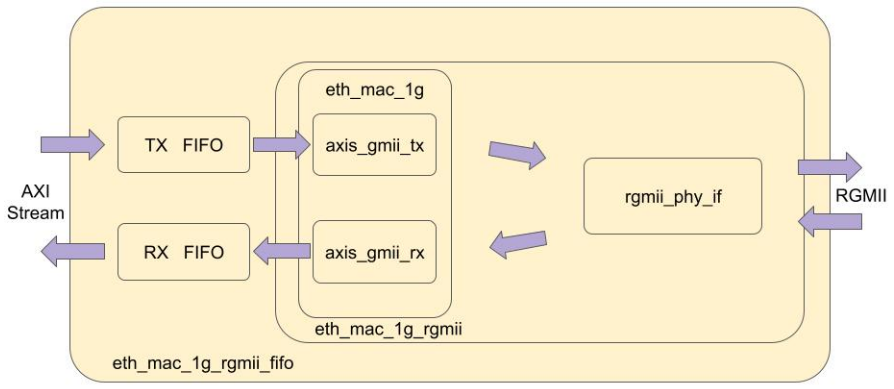


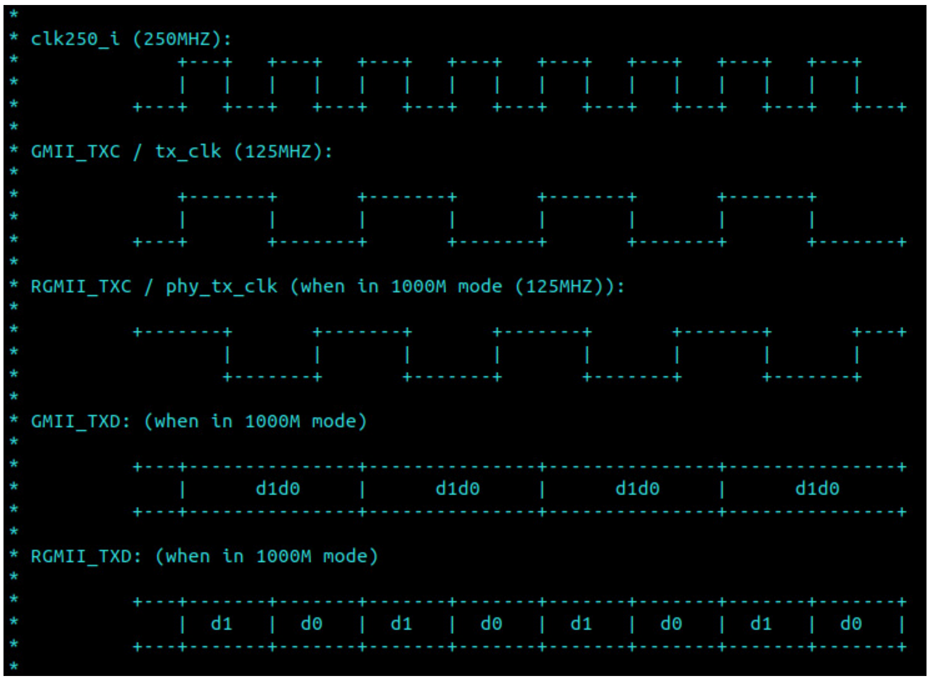
Figure 10: The eth_mac_1g_rgmii_fifo Module

Clock Domains

The clock domains of the MAC module are shown in figure 13. There are 3 clock domains in the design: logic_clk, clk250_i, and phy_rx_clk from the external PHY. clk250_i generates two additional clocks: tx_clk and phy_tx_clk. tx_clk is the clock used in the TX path of the MAC module, while phy_tx_clk is the RGMII TX clock sent to PHY. rx_clk is the output of a clock buffer for phy_rx_clk. It is used in the RX path of the MAC module.

Meeting the Timing Constraints

This paragraph describes how the RGMII timing constraints are met on Zedboard. First, the IOB packing has been used to help meet the TX RGMII timing. An IOB (Input Output Block) can hold a register and it is the interface between the FPGA and the outside world. What IOB packing does is it moves an external register into an IOB. This can minimize the skews between the RGMII TX clock and the RGMII TX data signals as both the clock signal and the data signals do not have to propagate for a long distance before reaching the I/O ports. Second, the IDELAYE2 primitives have been used to help meet the RX RGMII timing. An IDELAYE2 primitive simply delays input signals. Since the RX clock signal has a longer input path than that of the RX data signals (because the RX clock signal has to go through the clock buffer before reaching the registers), IDELAYE2 primitives are applied to the RX data signals to meet the RX timing.



```
*
* clk250_i (250MHZ):
*              +---+   +---+   +---+   +---+   +---+   +---+   +---+   +---+
*              |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
*          +---+   +---+   +---+   +---+   +---+   +---+   +---+   +---+   +---+
*
* GMII_TXC / tx_clk (125MHZ):
*
*              +-------+       +-------+       +-------+       +-------+
*              |       |       |       |       |       |       |       |
*          +---+       +-------+       +-------+       +-------+       +-------+
*
* RGMII_TXC / phy_tx_clk (when in 1000M mode (125MHZ)):
*
*          +-------+       +-------+       +-------+       +-------+       +---+
*                  |       |       |       |       |       |       |       |
*                  +-------+       +-------+       +-------+       +-------+
*
* GMII_TXD: (when in 1000M mode)
*
*          +---+---------------+---------------+---------------+---------------+
*              |     d1d0      |     d1d0      |     d1d0      |     d1d0
*          +---+---------------+---------------+---------------+---------------+
*
* RGMII_TXD: (when in 1000M mode)
*
*          +---+-------+-------+-------+-------+-------+-------+-------+-------+
*              |  d1   |  d0   |  d1   |  d0   |  d1   |  d0   |  d1   |  d0   |
*          +---+-------+-------+-------+-------+-------+-------+-------+-------+
*
```

Figure 11: Waveforms for rgmii_phy_if_waveform

3.2 RISC-V PLIC

3.2.1 Overview

RISC-V Platform-Level Interrupt Controller (PLIC)[1] multiplexes various interrupt sources onto interrupt lines of targets. Figure 14 shows the four main parts of the PLIC architecture: they are “interrupt signals”, “PLIC gateways”, “PLIC core” and “targets”.

“Interrupt signals” are the interrupt sources of PLIC, for example, they could come from an Ethernet device or a keyboard. A “PLIC gateway” converts its associated interrupt signal to an interrupt request, which then be forwarded to “PLIC core”, and the gateway sets the corresponding interrupt pending bit. One thing to note here is that a “PLIC gateway” only forwards one interrupt request at a time. It would only resume the forwarding after it gets an interrupt completion message for its associated interrupt source. “PLIC core” computes the EIP (External Interrupt Pending) bit and handles the accesses of the Interrupt Claim/Complete register for each target. Finally, a “target” would be signaled an interrupt notification if its corresponding EIP in “PLIC core” goes high. A “target” is usually a RISC-V hart context (figure 15). For example, a 4-core RISC-

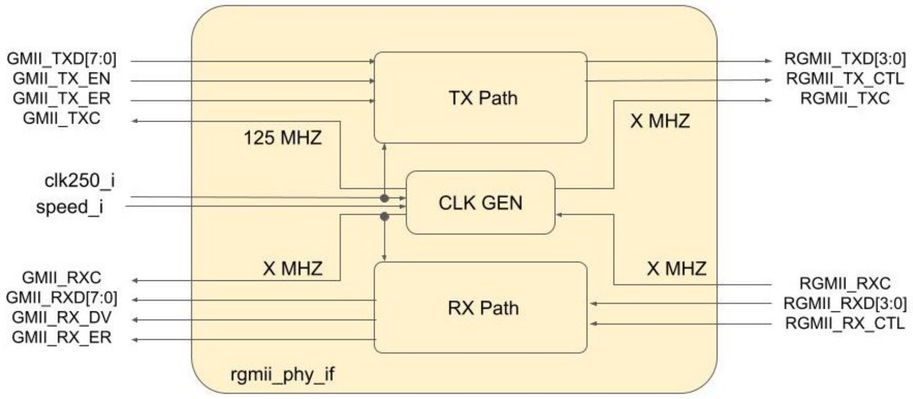


Figure 12: The rgmii_phy_if Module

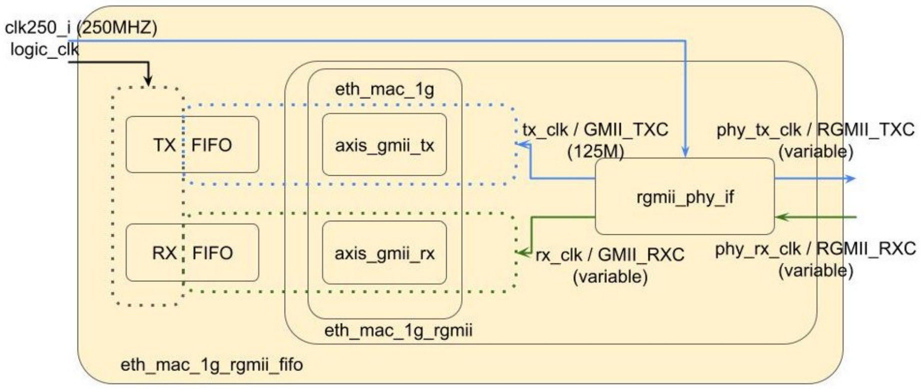


Figure 13: The Clock Domains of the MAC Module

V that supports M, S-mode could have up to 8 targets. However, in practice not all the hart contexts are connected to the PLIC. For instance, a typical RISC-V Linux system running in S-mode would only have the S-mode external interrupt lines of the RISC-V cores connected to the PLIC.

Besides the four main parts, there are also memory-mapped registers for controlling the PLIC operations. The “Interrupt Priorities” registers store the priority for each interrupt source, with 0 being disabled and higher value being higher priorities. The “Interrupt Pending Bits” registers show the pending status for each source. The “Interrupt Enables” registers store the enablement of every interrupt source of each target. The “Priority Thresholds” registers mask out all the interrupts with priorities less than or equal to it for each target. Lastly, the “claim/complete” registers are used to acquire the interrupt source ID of each target and send the interrupt completion message to the associated gateway.

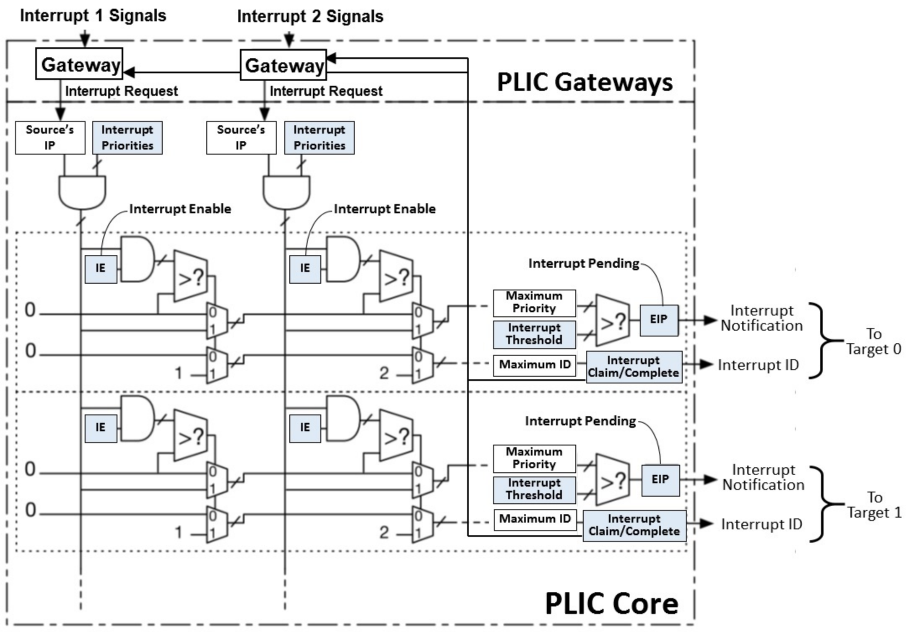


Figure 14: The PLIC Architecture [1]

3.2.2 Flow

Consider the scenario in figure 16, where the interrupt line of an Ethernet device and the interrupt line of a keyboard are connected to PLIC, which is then connected to the S-mode external interrupt lines of the RISC-V cores. Assuming the priorities of these two interrupt sources are higher than the two thresholds for the RISC-V cores, and assuming these two interrupt sources are all enabled for each RISC-V core, below demonstrates the interrupt flow when the Ethernet device start asserting its interrupt line while the keyboard interrupt line is deasserted: First, the gateway of interrupt source 1 forwards the Ethernet interrupt to the PLIC core and delays its next forwarding until receiving an interrupt completion message. This corresponds to step 1 and 2 in figure 17. Second, since interrupt source 1 is enabled for both RISC-V cores, the PLIC core sends out an interrupt notification to each S-mode interrupt line. This is step 3.

Third, both RISC-V cores get the S-mode external interrupt and start executing their interrupt handlers. At the beginning of their handlers, both RISC-V cores read their own

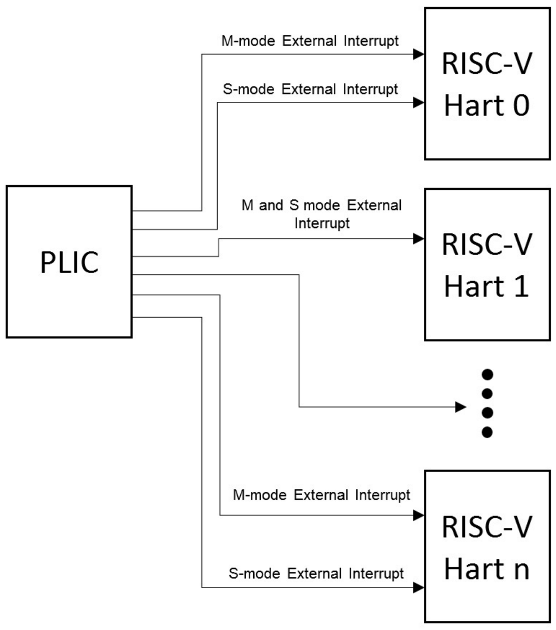

M-mode External Interrupt
S-mode External Interrupt
RISC-V Hart 0
PLIC
M and S mode External Interrupt
RISC-V Hart 1
M-mode External Interrupt
S-mode External Interrupt
RISC-V Hart n

Figure 15: PLIC and RISC-V Harts [1]

memory-mapped claim register in PLIC. If the read returns a non-zero value, it means the core that did the read has successfully claimed the interrupt and got the ID of the interrupt, in this case, 1. Otherwise, it means the interrupt has been claimed by another core. This ensures only one core claims an interrupt at a time. After the claim, the interrupt pending bit for source 1 is cleared, and since there's no other pending interrupt, both EIP bits are cleared as well, resulting in a clear in each S-mode interrupt line. The above process corresponds to step 4 and 5. After handling the Ethernet interrupt, the RISC-V core that claimed the interrupt writes the interrupt ID, 1, back to its memory-mapped completion register in PLIC. This action sends a completion message to the gateway of interrupt source 1, enabling it to forward the next interrupt request. This completes step 6.

3.2.3 RISC-V PLIC from OpenTitan

A RISC-V PLIC module from lowRISC has been used in BlackParrot. The PLIC module almost works out-of-box on BlackParrot. Only a few custom changes have been applied to the module, for example, to workaround a synthesis bug appearing in some older Vivado version. The PLIC module was generated from the PLIC template in OpenTitan.

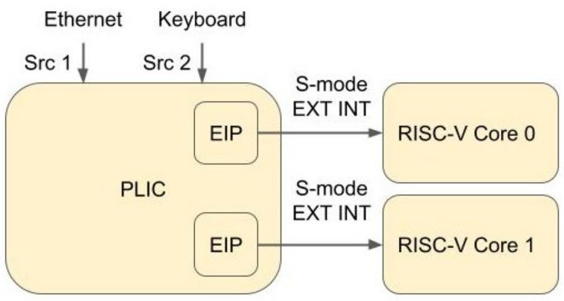


Figure 16: The PLIC Example

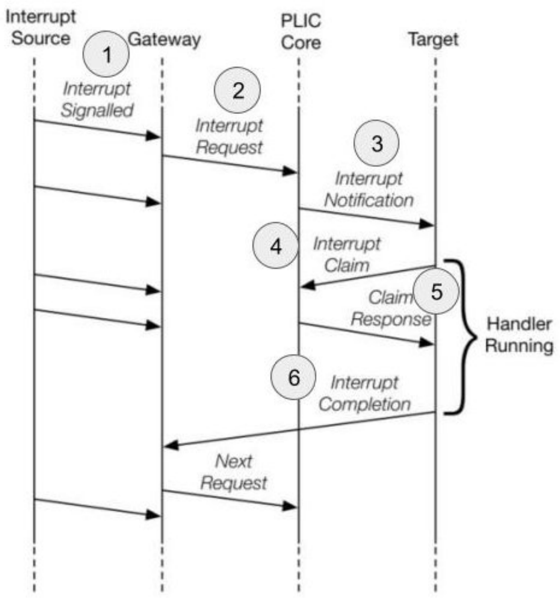


Figure 17: The PLIC Interrupt Flow [1]

Although the PLIC template can support up to 255 interrupt sources, for the system of BlackParrot plus Ethernet controller, only two interrupt sources (Note that one of the two interrupt sources is reserved and not used, that is, source 0) and one target are generated.

4 Software

4.1 Linux

4.1.1 Linux Liteeth Driver

Memory Map

Figure 18 shows the memory map of a Liteeth device. The map consists of two parts, the TX/RX buffer region (starting from buf_base) and the register region (starting from reg_base). Note that in Liteeth driver, WRITER refers to the RX side of the Liteeth hardware; while READER refers to the TX side. Also, buf_base is not necessarily lower than reg_base. They can be any arbitrary locations.

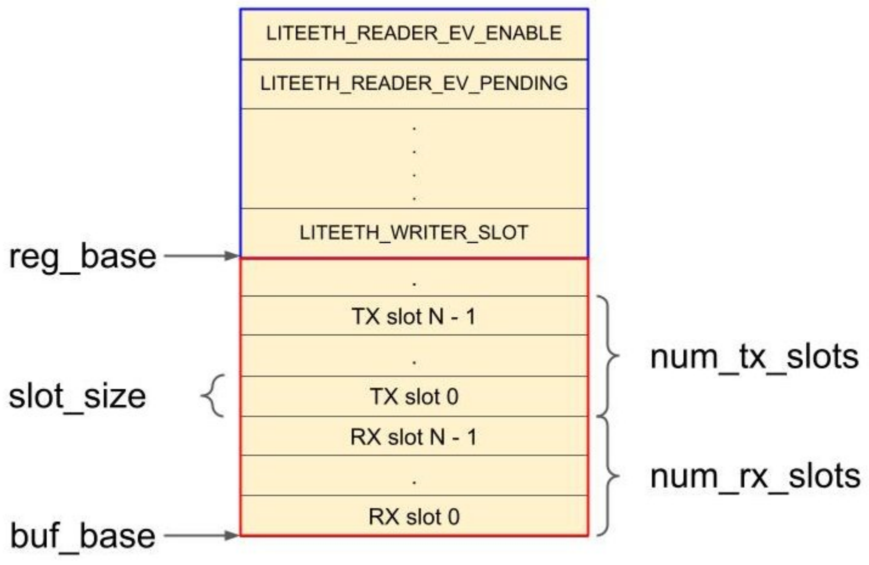


Figure 18: The Liteeth Memory Map

- The TX/RX Buffer Region

  This region consists of a TX FIFO and an RX FIFO. They contain num_tx_slots and num_rx_slots slots respectively. Each slot has the same size and is able to store 1 packet. The current RX slot pointer is maintained in the hardware and can be read out through LITEETH_WRITER_SLOT. On the contrary, the current TX slot pointer is maintained in the driver, and needs to be written to LITEETH_READER_SLOT to inform the hardware the right location to send the packet.

- Memory-Mapped Registers

  Below lists only part of the memory-mapped registers that are used in the driver.

  - LITEETH_WRITER_SLOT

    Index of the current received packet.
  - LITEETH_WRITER_LENGTH

    Length of the current received packet.
  - LITEETH_WRITER_EV_PENDING

    Pending bit for RX. If set, the RX buffer is non-empty. Writing 1 to it clears the bit.
  - LITEETH_WRITER_EV_ENABLE

    Enable bit for RX. If set, the interrupt line of RX is asserted when LITEETH_WRITER_EV_PENDING is set.
  - LITEETH_READER_START

    If written to 1, the next packet in the TX buffer is transmitted.
  - LITEETH_READER_READY

    If set, the TX buffer is full.
  - LITEETH_READER_SLOT

    Index of the next transmitting packet.
  - LITEETH_READER_LENGTH

    Length of the next transmitting packet.
  - LITEETH_READER_EV_PENDING

    Pending bit for TX. If set, the TX buffer becomes free. Writing 1 to it clears the bit.
  - LITEETH_READER_EV_ENABLE

    Enable bit for TX. If set, the interrupt line of TX is asserted when LITEETH_READER_EV_PENDING is set.

Functions Below lists the main functions used in the driver.

- liteeth_rx

  The receive function for Liteeth. It reads out the value from LITEETH_WRITER_SLOT to get the memory location for receiving the packet. It also reads out LITEETH_WRITER_LENGTH to get the packet size. If the packet size is too large or is zero, it drops the packet; otherwise the packet is copied into the Linux kernel from the RX buffer in the Ethernet hardware, followed by an ACK. One thing to note is that this function receives only one packet per call. This might be a suboptimal solution because the CPU will have to take the interrupt latency for every packet.

- liteeth_interrupt

  The interrupt routine checks TX and RX pending bits, and clears them if set. If the TX pending bit is set, it resumes the network traffic in the Linux kernel if it has been stopped. If the RX pending bit is set, it calls liteeth_rx to receive the packet.

- liteeth_probe

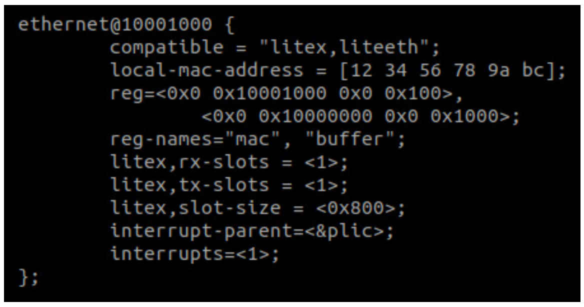
  This function is called when the Liteeth driver is registered in the Linux kernel. It parses the device node with 'compatible' property set to "litex,liteeth" in the device tree and determines the parameters in figure 18. Figure 19 shows an example device tree configuration for the Liteeth driver. The "regs" specifies the bases and the sizes for the TX/RX region and register region (in this case, reg_base is set to 0x10001000 with size 0x100; buf_base is set to 0x10000000 with size 0x1000); the "litex,rx-slots" and "litex,tx-slots" specify num_rx_slot and num_tx_slot, respectively; and the "litex,slot-size" specifies slot_size.

- liteeth_start_xmit

  This function is called when a packet needs to be sent. If the TX buffer is full before sending the packet, it tells the Linux kernel to stop transmitting packets. This stop serves as a means of network flow control, and it will remain in effect until Linux receives the interrupt indicating that the TX buffer has become available. If the transmitting packet size is too large, the packet gets dropped. Otherwise the function writes the packet to the TX buffer, the packet size to LITEETH_READER_LENGTH, the correct TX slot index to LITEETH_READER_SLOT, and then issues a start signal to LITEETH_READER_START. At the end, the driver updates the TX slot index.

- liteeth_open, liteeth_stop, liteeth_remove

  These functions are part of the net device operations in the Linux kernel.



```
ethernet@10001000 {
        compatible = "litex,liteeth";
        local-mac-address = [12 34 56 78 9a bc];
        reg=<0x0 0x10001000 0x0 0x100>,
                <0x0 0x10000000 0x0 0x1000>;
        reg-names="mac", "buffer";
        litex,rx-slots = <1>;
        litex,tx-slots = <1>;
        litex,slot-size = <0x800>;
        interrupt-parent=<&plic>;
        interrupts=<1>;
};
```

Figure 19: A Device Tree Example for the Liteeth Driver

4.1.2 Linux Interrupt Controller Driver

There are two RISC-V interrupt controller drivers in Linux kernel 5.15: the RISC-V Hart-Level Interrupt Controller[26] and the SiFive Platform-Level Interrupt Controller[12]. **RISC-V Hart-Level Interrupt Controller (HLIC)**

This controller handles the interrupts defined by the RISC-V specification: they are the software interrupt, timer interrupt and external interrupt. The use of this interrupt controller is specified in the device tree. The device node for HLIC should have the 'compatible' property set to "riscv,cpu-intc". As for connecting the interrupt lines of a device to the interrupt controller, "interrupts-extended" can be used to specify those connections. For example, the example device tree in figure 20 shows a clint device connecting its M-mode software interrupt line (Number: 3) and M-mode timer interrupt line (Number 7) to a Hart-Level Interrupt Controller, called CPU0_intc.

When Linux registers an HLIC, the callback function, handle_arch_irq, within handle_exception (which is the entry point of the S-mode interrupt handling in Linux), is set to riscv_intc_irq, which is the interrupt handler in this controller, as demonstrated in figure 21. Besides, Linux will create an irq_domain data structure to keep the mapping between the hardware irq numbers (like 9 is S-mode external interrupt in a RISC-V hart) and their associated interrupt handlers during the HLIC registration. Linux will add entries to that data structure whenever it discovers a device connecting to the HLIC in the device tree. For instance, there is an HLIC, cpu0_intc in figure 20, therefore an irq_domainfor the HLIC is created. There is also a PLIC device connected to cpu0_intc, so an interrupt handler provided by the PLIC device and the hardware irq number 9, are put into the irq_domain of cpu0_intc, as shown in figure 21. With this mapping, HLIC is able to find out the correct interrupt handler to call based on the passed hardware irq number.

SiFive Platform-Level Interrupt Controller (PLIC)

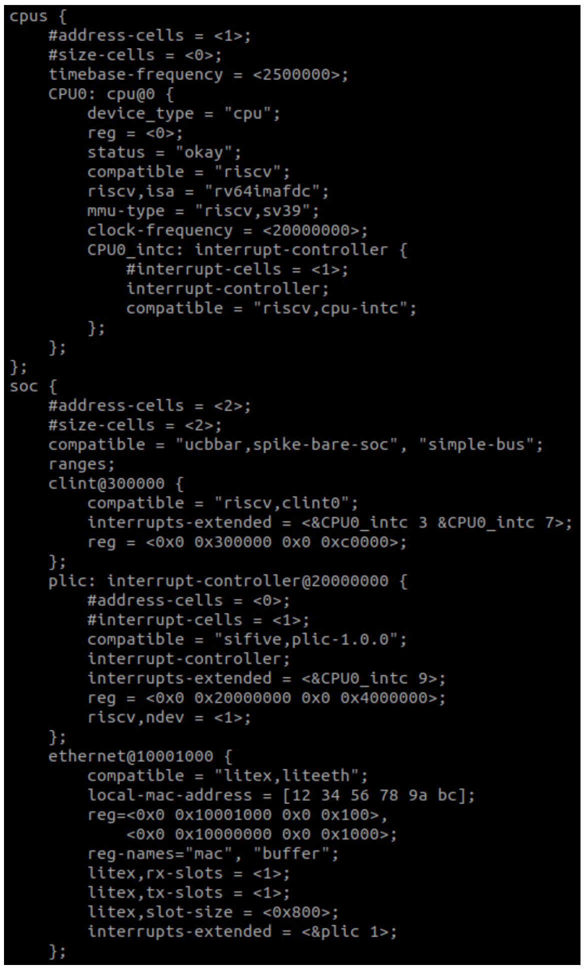
While HLIC handles the three standard RISC-V interrupts, this controller deals with the interrupts connected to a PLIC device. The use of this interrupt controller is specified in the device tree. The device node for PLIC should have the 'compatible' property set to "sifive,plic-1.0.0". The 'extended-interrupt' in figure 20 shows a PLIC interrupt controller, plic, which connects to the S-mode external interrupt line (number 9) of an HLIC interrupt controller, CPU0_intc. The 'reg' property defines the location and the size of the memory-mapped IO of the PLIC device, in this case they are 0x20000000 and 0x4000000. Finally, the 'riscv,ndev' specifies the total number of interrupts supported in the PLIC device.

When Linux registers a PLIC, it not only creates an irq_domain for it, but also adds the mapping between the interrupt handler of PLIC, plic_handle_irq, and the hardware irq number to the irq_domain of its interrupt parent domain, in this case the irq_domain for CPU0_intc. What plic_handle_irq does is that it reads the claim register in the PLIC device to get the interrupt source number. Then it looks up the number in the plic's irq_domain to find out the right interrupt handler to call.

Going Back to the device tree in figure 20, there is a Liteeth Ethernet device, ethernet, connected to the interrupt source 1 of plic, so the mapping between the Liteeth interrupt handler, liteeth_interrupt, and 1 is added to plic's irq_domain. Note, however, instead of being called directly by plic_handle_irq, liteeth_interrupt is called by handle_fast_eoi, which sends an End of Interrupt signal to PIC (essnetially a write to the PLIC completion register) after the liteeth_interrupt call. This completes the PLIC interrupt handling.

Why not connecting Liteeth Driver to HLIC

Possibly the simplest way of combining a Liteeth Ethernet controller with a RISC-V core



```
cpus {
    #address-cells = <1>;
    #size-cells = <0>;
    timebase-frequency = <2500000>;
    CPU0: cpu@0 {
        device_type = "cpu";
        reg = <0>;
        status = "okay";
        compatible = "riscv";
        riscv,isa = "rv64imafdc";
        mmu-type = "riscv,sv39";
        clock-frequency = <20000000>;
        CPU0_intc: interrupt-controller {
            #interrupt-cells = <1>;
            interrupt-controller;
            compatible = "riscv,cpu-intc";
        };
    };
};
soc {
    #address-cells = <2>;
    #size-cells = <2>;
    compatible = "ucbbar,spike-bare-soc", "simple-bus";
    ranges;
    clint@300000 {
        compatible = "riscv,clint0";
        interrupts-extended = <&CPU0_intc 3 &CPU0_intc 7>;
        reg = <0x0 0x300000 0x0 0xc0000>;
    };
    plic: interrupt-controller@20000000 {
        #address-cells = <0>;
        #interrupt-cells = <1>;
        compatible = "sifive,plic-1.0.0";
        interrupt-controller;
        interrupts-extended = <&CPU0_intc 9>;
        reg = <0x0 0x20000000 0x0 0x4000000>;
        riscv,ndev = <1>;
    };
    ethernet@10001000 {
        compatible = "litex,liteeth";
        local-mac-address = [12 34 56 78 9a bc];
        reg=<0x0 0x10001000 0x0 0x100>,
            <0x0 0x10000000 0x0 0x1000>;
        reg-names="mac", "buffer";
        litex,rx-slots = <1>;
        litex,tx-slots = <1>;
        litex,slot-size = <0x800>;
        interrupts-extended = <&plic 1>;
    };
```

Figure 20: A Device Tree Example for the RISC-V Interrupt Controllers

is to just bind the interrupt line of the Ethernet controller directly to an external interrupt line of the core. From the hardware perspective, this is doable if the Ethernet controller is the only external device in the system. However, in Linux, the Liteeth driver cannot work well with the HLIC driver.

When the HLIC driver gets registered, an irq_domain is created for it. One of the domain operations (to be specific, the domain map operation) of HLIC's irq_domain will only allow interrupt handlers that are marked as 'percpu' to be mapped and added to its irq_domain. Since the interrupt handler from Liteeth (liteeth_interrupt) is not 'percpu', the registration of the Liteeth device will fail. The PLIC drive, on the other hand, does not have such an issue because its interrupt handler is 'percpu' and its irq_domain allows interrupt handlers without 'percpu' to be added to its irq_domain. Therefore, from the

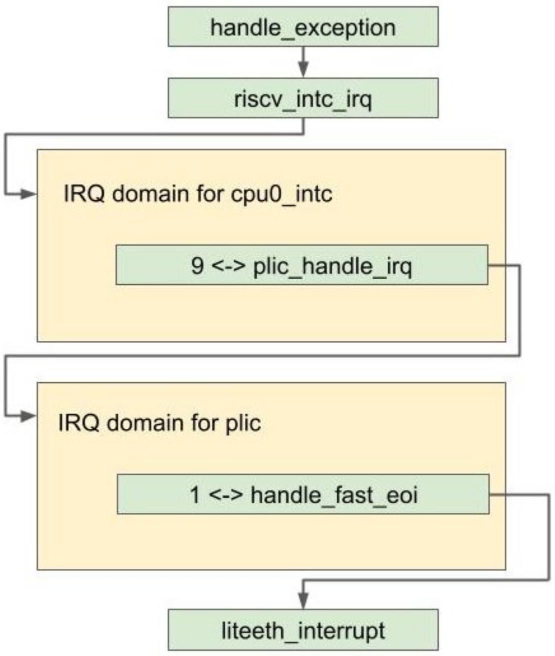


Figure 21: The Linux IRQ Domains for HLIC and PLIC

software perspective, the Liteeth driver needs to connect to the PLIC driver first, and only with that the PLIC driver can then connect to the HLIC driver. This is one of the reasons why a PLIC device is necessary in this thesis work if one of the goals is to use everything as is.

Percpu Interrupt Handler

A 'percpu' interrupt handler can be called on different CPUs at the same time, without the need of any locking mechanism. Unlike the typical interrupt handler, 'percpu' interrupt handler has per-cpu storage. In other words, each CPU core will have its own copy of data. When any of the CPUs executes a 'percpu' interrupt handler, it will first figure out what its CPU ID is and it will use that ID to get the location of its per-cpu storage, on which the rest of the handler will then operate. Therefore no race condition will occur in a 'percpu' interrupt handler.

'percpu' interrupt handlers handle CPU-related tasks. For example, the local timer interrupt handler increments the tick counts of the running core; the PLIC interrupt handler accesses the PLIC memory-mapped registers of the running core. In contrast, 'percpu' interrupt handlers are not suitable for tasks such as receiving Ethernet packets, as it is

CPU-agnostic and does not behave differently when running on different CPUs.

4.1.3 The RX/TX Path in Linux

The RX Path

After the Ethernet controller receives an Ethernet packet, it asserts the interrupt line to PLIC, which then sends an interrupt notification onto the S-mode external interrupt line of BlackParrot. After getting the notification, BlackParrot is trapped into the S-mode external interrupt handler, handle_exception, in Linux (because opensbi has delegated the S-mode external interrupt to Linux), which then calls riscv_intc_irq with interrupt number 9. riscv_int_irq looks up the number in its irq_domain structure and finds out the corresponding handler, plic_handle_irq. plic_handle_irq gets called and reads the claim register in PLIC and gets the interrupt source number, which is 1 in this case. The handler then again looks up the number in its irq_domain and finds out the corresponding handler, which is handle_fasteoi_irq. This function gets called and then it calls liteeth_interrupt, which is the interrupt handling process for the Ethernet. After liteeth_interrupt is done, handle_fasteoi_irq sends an EOI (essentially a write to the completion register) to PLIC and ends the entire receive flow.

The TX Path

Whenever the Linux kernel gets a packet to transmit for the Liteeth driver, it calls liteeth_start_xmit. Note the transmission can be stopped temporarily if the kernel previously found that the TX buffer from the Ethernet controller was full during its last transmission. The kernel will resume the transmission after getting an interrupt indicating that the TX buffer has become available.

4.2 lwIP

4.2.1 Scope

This work targets embedded applications with small memory/code footprint without operating systems. The final work should be able to run on the BlackParrot Ethernet controller and the RISC-V PLIC module, and support TCP, UDP and IPv4.

4.2.2 Introduction

lwIP is a lightweight implementation of many well-known network protocols in layer 3 and 4, such as TCP, UDP, IP(both version 4 and 6) and DHCP. lwIP also provides device drivers for some common layer 2 protocols, for example, the Ethernet protocol. Figure 22 shows the overview of an lwIP application running on the Ethernet: lwIP is located in layer 3, 4 of the OSI model, and the device driver serves as an interface between lwIP and the underlying physical media.

Depending on the use cases, lwIP can either run in the OS mode (with an operating system) or in the mainloop mode (without an OS). The OS mode supports sequential API that is similar to the Linux socket API, while the mainloop mode supports raw API in which the program execution is driven by callback functions. Since my work targets embedded applications without operating systems, the mainloop mode is the main focus in this thesis work.

One of the major differences between the mainloop mode and the OS mode is the preemption of the task. In the OS mode, the tasks spawned by the user program do not have to yield themselves to the other tasks because the OS will take timer interrupts to decide if a task should be preempted and replaced with another task. On the other hand, in the mainloop mode, the tasks have to yield themselves from time to time in order not to block the entire system. This would make programming in the mainloop mode more challenging. Figure 23 shows the typical programming strategy for the mainloop mode. After the system initialization, the program enters an infinite loop and executes each task in order. Tasks are implemented as state machines so that all tasks are able to resume their previous work in every iteration.

4.2.3 Components

Figure 24 shows the components of lwIP. TCP and UDP connections are described by tcp_pcb structs and udp_pcb structs, respectively. Each lwIP network interface, which is described by a netif struct, has its own IP address. Based on the destination IP address, the TCP / UDP traffic is routed to an interface by the IP module. Each interface also has its

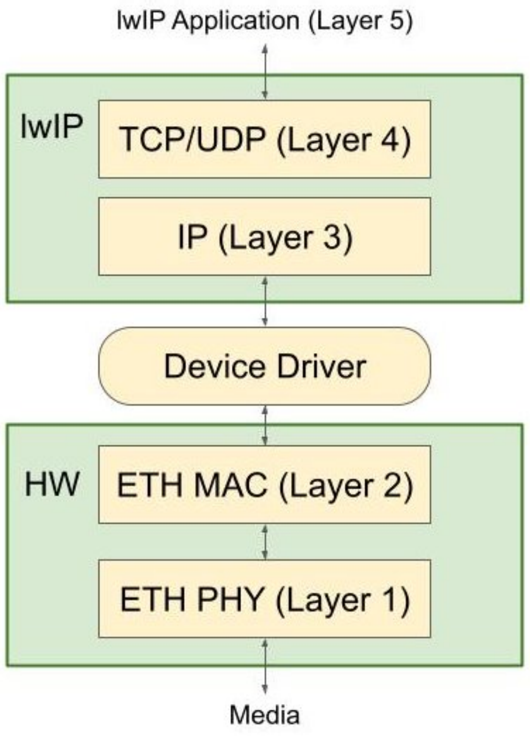
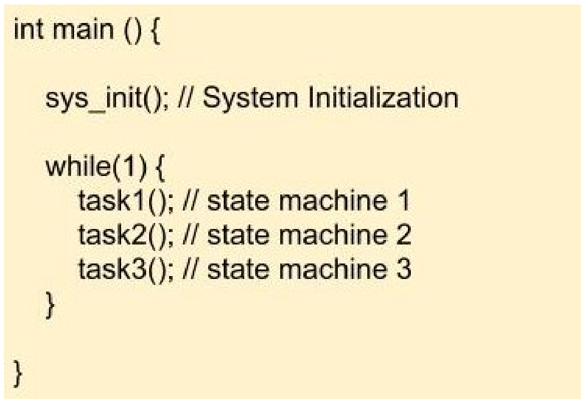


Figure 22: lwIP Overview

own configurations, for example, the MTU size, the hardware address of the underlying hardware, and functions used by its associated device driver to send / receive the packets to / from lwIP. The boundary between lwIP and the device driver is *netif->input* and *netif->output* (in the case of IPv4). *netif->input* is called by the driver to send packets in lwIP, while *netif->output* is called by the interface to send packets out lwIP. Although not shown in the figure, lwIP uses pbuf structs to pass the packet/payload between different components. Each pbuf can point to one chunk of memory. Multiple pbufs can form a singly-linked list to store a piece of information that are not contiguous in the memory.

4.2.4 Flow

lwIP uses callback-style API in the mainloop mode to notify the application of the happening events. The below pseudo code gives the basic idea of how the API works. Before entering the main loop, the program registers callback functions for different events, such as the TCP data received event or the error notification event. In the main loop, the program does the following three things: 1. Check if there is any received



```
int main () {

   sys_init(); // System Initialization

   while(1) {
      task1(); // state machine 1
      task2(); // state machine 2
      task3(); // state machine 3
   }

}
```

Figure 23: Mainloop

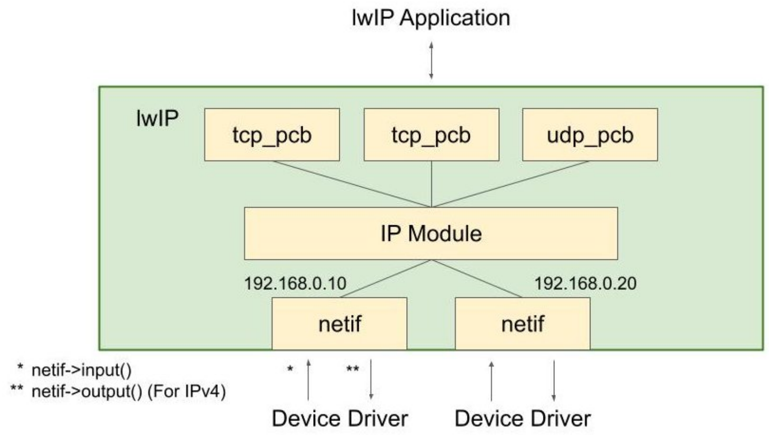


Figure 24: lwIP Components

packet. If yes, send it to lwIP through a netif. 2. Call the lwIP housekeeping function, *sys_check_timeouts*, to take care of all the background protocol operations 3. Call the user task. *sys_check_timeouts* checks the current time and calls a timeout handler if any timer for a network protocol expires, for example, resending a TCP packet when lwIP does not receive an ACK in time. *sys_check_timeouts* should be called from time to time from the main loop, otherwise lwIP will not be able to handle the expired timeout handler properly.

```
void callback1(void);
void callback2(void);
void callback3(void);

struct netif netif;

int main() {
    lwip_register_event1_callback(callback1);
    lwip_register_event2_callback(callback2);
    lwip_register_event3_callback(callback3);

    while(1) {
        receive_packet(&netif);
        sys_check_timeouts(); // the lwIP housekeeping function
        user_task();
    }
}
```

4.2.5 Dynamic Memory Allocation

lwIP has two dynamic memory allocators, which are the heap memory allocator and the memory-pool allocator. Their corresponding functions are *mem_malloc/mem_free* and *memp_malloc/memp_free*. Both the heap memory and the memory pools are simply statically-allocated memory and their total sizes can be specified in the lwIP configuration.

An lwIP memory pool is a chunk of memory that consists of several memory blocks, each of which has the same size. Figure 25 shows an example of memory pools in lwIP. The yellow bars are the memory pools, and the green boxes are the descriptors for the memory pools. Using the lwIP memory pool allocator has two advantages: 1. The allocation and free operation takes only constant time, since each free memory block in a memory pool is put into the head of a singly-linked list. 2. It avoids memory fragmentation, because each memory pool only holds objects with the same size.

For *mem_malloc/mem_free*, lwIP provides three implementations. The first one (the default one) allocates various sizes of memory chunks from the lwIP heap. The second one simply calls an external memory allocator provided by the user, which can reduce code size if there is already a memory allocator in the user's library(like *malloc* from newlib). The third one uses the *memp_malloc/memp_free* as its backend. It will search from the memory pool with the smallest block size and return the allocated memory once it finds a block that is just big enough for the user's memory request.

For *memp_malloc/memp_free*, they can be configured to allocate from the lwIP memory pools or allocate memory with *mem_malloc/mem_free*. The latter is useful when *mem_malloc/mem_free* are also configured to the second implementation mentioned in the previous paragraph, as the entire program will then only have the user-provided memory allocator, which could save some code footprint.

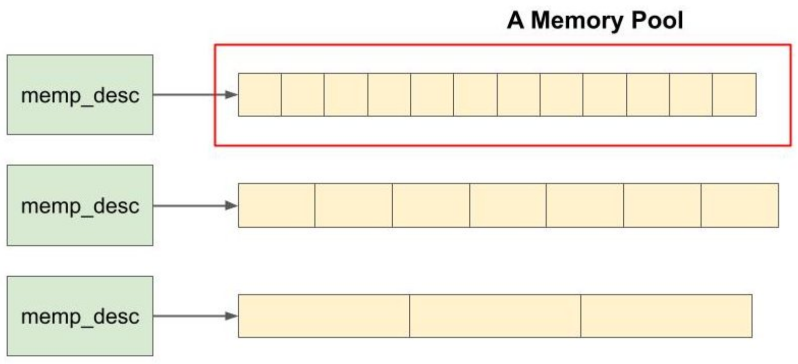


Figure 25: lwIP Memory Pools

4.2.6 Porting

The actual porting process for lwIP depends on several factors, including the toolchain being used, the underlying networking device, and the features that users would like to have. For the toolchain and the networking device, the GNU C compiler, the Newlib C library, and an Ethernet controller have already been ported to BlackParrot. For the lwIP features, there will be no OS in our applications so only the mainloop mode is considered. Below gives the list of what has been done for the porting to the BlackParrot platform:

- Implement *sys_now*

  This function is necessary for the timeout features in lwIP to work. In the case of BlackParrot, *gettimeofday* from BlackParrot Newlib is used to implement this function.

- Create arch/cc.h

  If users would like to overwrite the default settings in lwip/arch.h, which defines macros for different processors and compilers, they should provide their own macros in this user provided file, arch/cc.h, which is included at the very beginning of lwip/arch.h. Below list some examples of the macros defined in lwip/arch.h: macro that specifies the byte order of the processor (little endian), macro that defines the platform specific diagnostic output function (printf), and macros that specify the mappings between the lwIP's data types to the compiler's data types.

  In the case of BlackParrot, the macro LWIP_DECLARE_MEMORY_ALIGNED has been redefined in this file to leverage the alignment attribute from the GNU C compiler to declare an aligned byte array.

- Create lwip/lwipopts.h

  If users would like to overwrite the default settings in lwip/opt.h, which defines various kinds of features options, such as options for TCP, UDP, IP protocols, options for memory allocation, and options for debugging and statistics, they should provide their own options in this file.

  lwip/lwipopts.h plays a crucial role in lwIP tuning, and the actual settings in this file highly depend on the actual program to be run. For example, if the program needs high performance, options that tell lwIP to use memory pool allocation in *mem_malloc* can be put in this file; on the other hand, if the program needs ultra low memory footprint, options that disable unused features/modules and options that allocate fewer memory can also be put in this file. For tuning the memory footprint of the lwIP memory allocators, MEM_STATS can be set to 1 in lwipopts.h and the function *stats_display* can be used to dump the memory usage at any specific moment during runtime.

- Implement the device drivers for the BlackParrot Ethernet device

  Each network interface in lwIP is described by a netif structure, which includes the following three callbacks:

  - input: send the received packet to lwIP.
  - output: resolve the hardware address of the IPv4 packet and send the packet out to the network device.
  - linkoutput: send out the layer-2 packet to the network device. This callback only needs to be set when output requires it.

  The above callbacks are the boundary between lwIP and the device driver. In the case of Ethernet, lwIP already has built-in functions, netif_input and etharp_output, for input and output respectively, so that the user does not need to re-implement all the callbacks. The only callback the user needs to implement is linkoutput since this will be called within etharp_output. Besides these callbacks, the user also has to implement their device driver which will communicate with the hardware and interact with the lwIP interface with the above callbacks. For BlackParrot, the built-in Ethernet functions mentioned above are used and linkoutput is set to one of the BlackParrot Ethernet driver functions, ethernet_send, which will appear in section 4.4.2.

Note that in the mainloop mode, the porting of synchronization primitives such as semaphores and mutexes is not required as all the tasks will not be preempted by each other, and there can only be one task active at a time. Although the interrupt handler (described in section 4.4.2) that receives the packet can preempt a task, a simple lock-free mechanism has been used to allow safe concurrent data access.

4.3 wolfSSL

4.3.1 Introduction

wolfSSL is a lightweight TLS library suitable for embedded systems. It supports up to TLS 1.3, which is the latest TLS version. Although nowadays TLS and SSL are often used interchangeably to mean the TLS protocol, they are actually two different protocols: For TLS, there are TLS 1.0 to TLS 1.3, whereas for SSL, there are SSL 1.0 to SSL 3.0, which have already been replaced by TLS. In this thesis, these two terms will continue to be used interchangeably to mean the TLS protocol.

Like TCP, an SSL client will initiate a handshake at the beginning of an SSL connection to an SSL server. One of the steps happening in the SSL handshake is authenticating the server's identity. The main purpose of having the server authentication is to make sure that the client is really connecting to the server that says who it is. Without the authentication, the server can claim to be anyone on the Internet without being noticed by the client.

The authentication process involves an important piece of data called, certificate. Figure 26 shows how a certificate is generated and used in an SSL connection. A certificate, issued by a Certificate Authority (CA), contains the owner's public key and the owner's identity. The issue of a certificate can be described with two steps: First, CA receives a certificate signing request (CSR) from the organization that plans to have the certificate. A CSR includes information like the legal name, email address and the physical address (city, state, country...) of the organization, and a public key. Second, after CA verifies the information on the CSR, it creates a certificate and signs it with its private key. The above steps are how the server and CA certificate are made.

With the server certificate, the server can now show its identity by sending its certificate to its client during the SSL handshaking. After receiving the server certificate, the client starts the server authentication: First, it figures out which CA signed the server certificate. Then it searches its trusted CA certificate list (pre-installed in the client) to find the CA's public key. Finally, the client uses the CA's public key to verify if the server certificate has really been verified and signed by the CA mentioned in the server certificate. If yes, then the server certificate certifies the mapping between the owner's public key and the owner's identity.

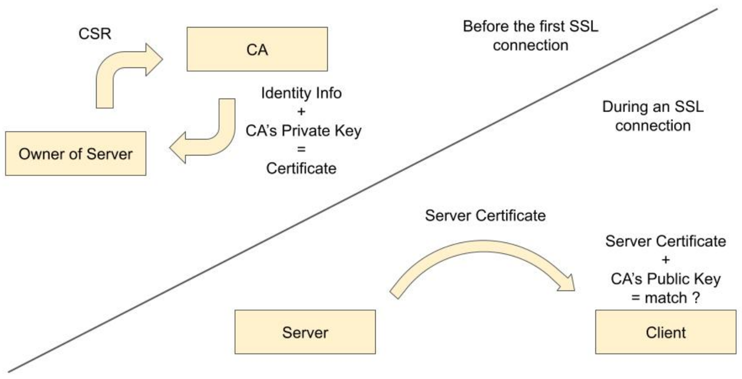

CSR
CA
Before the first SSL connection
Identity Info + CA's Private Key = Certificate
During an SSL connection
Owner of Server
Server Certificate
Server Certificate + CA's Public Key = match ?
Server
Client

Figure 26: How an SSL Certificate Works

4.3.2 Porting

The actual porting of wolfSSL depends on the use case and the available software/hardware resources. For BlackParrot, BlackParrot Newlib and the lwIP library are available. This section describes the actions taken to port wolfSSL to BlackParrot with those libraries.

- WRITEV

  By default, wolfSSL includes <sys/uio.h> and provides *wolfSSL_writev* to the program. Since the header is not supported in BlackParrot Newlib, this feature is disabled by defining NO_WRITEV.

- INPUT / OUTPUT

  wolfSSL uses the callbacks CBIORecv and CBIOSend to interact with the underlying transport layer. By default, those two callbacks are set to *EmbedReceive* and *EmbedSend*, which defaults to using the *recv* and *send* from a BSD socket interface. In order to integrate lwIP into wolfSSL, WOLFSSL_USER_IO is defined and the callbacks CBIORecv, CBIOSend are set to *ssl_media_send* and *ssl_media_recv*, which interact with lwIP.

- FILESYSTEM

  Although there is lightweight filesystem support in BlackParrot Newlib, the filesystem support in wolfSSL is disabled by passing –disable-filesystem to ./configure to achieve a smaller code size.

- TIME

  Although BlackParrot Newlib already has a time function, *gettimeofday*, some modification to the time function is still needed for wolfSSL. In SSL, current time is used to validate certificates: Each certificate has notBefore and notAfter fields, and the current time has to fall within this time window to pass the validation. By the time the wolfSSL was porting to BlackParrot, there was no real time clock in BlackParrot that could keep the current time. Therefore, in order to work around this timing issue, the *gettimeofday* implementation in BlackParrot Newlib was changed to return an Epoch time that starts from a fixed time base, for example, June 1, 2021 12:00:00 AM. The expiration date of the demo certificates (which is located in the folder tcp_ssl/ssl_cert_build/. This can be found in section 4.4.1) were set accordingly.

- RANDOM SEED

  By default, wolfSSL uses /dev/urandom or /dev/random to retrieve random numbers. Since no such device files exist in our environment, NO_DEV_RANDOM is defined and the *rand* function from BlackParrot Newlib is used instead.

- THREADING
  Since there will only be one task active in the system, and tasks will not preempt with each other, –enable-singlethreaded is passed to ./configure to negate the need to port the wolfSSL mutex layer.

4.4 Ethernet Examples

4.4.1 Directory Structure

Figure 28 shows the directory structure overview. examples/ includes three examples, tcp_ssl/, ssl_loader/, tftp_loader/, and a common folder common/, which includes the Ethernet device driver API for all the examples. tcp_ssl/ contains TCP server/client programs and SSL server/client programs. ssl_loader/ and tftp_loader/ contain the boot loaders for the BlackParrot System on Module, which is described in section 4.4.4.

common/ethernet_driver/ has ethernet_driver.c, which provides the standard Ethernet driver interface for lwIP, and three sub-folders, interrupt/, polling/ and host/, which provide the actual driver implementations. interrupt/ uses interrupts to receive packets. polling/ uses simple polling to receive packets. host/ uses the raw Linux socket API as the means to send and receive packets through a real Ethernet interface on a host computer. The idea behind host is that, with host, developers can choose to compile the programs using the host toolchain and run them natively on a host. This opens up several opportunities, for example, improving the debugging process, as developers can run dynamic analysis tools like Valgrind to catch memory bugs (use-after-free, memory leaks), or leveraging the profiling tools in the host to identify any potential bottlenecks in the program.

Each example folder comes with two sub-folders: lwip/ and main/. lwip/ contains the code that glues the lwIP network stack and the application code in main/ together, while main/ contains all the application code that sits in the upper layer of the Internet (figure 27), for example, the TLS/SSL layer. These two sub-folders are designed to be independent modules, so users can easily replace any part of the design if needed.

Besides examples, there are also dromajo_config/ and import/. dromajo_config/ contains patches that add a PLIC device and a BlackParrot Ethernet Controller to BlackParrot Dromajo, so that developers can run and debug the examples on it. However, only examples with the polling driver implementation will work, as BlackParrot Dromajo cannot handle external interrupts properly at the time of writing this thesis. import/ contains all the submodules: BlackParrot Dromajo, lwIP and wolfSSL.

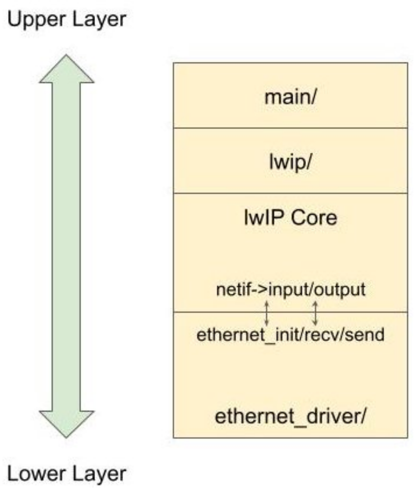
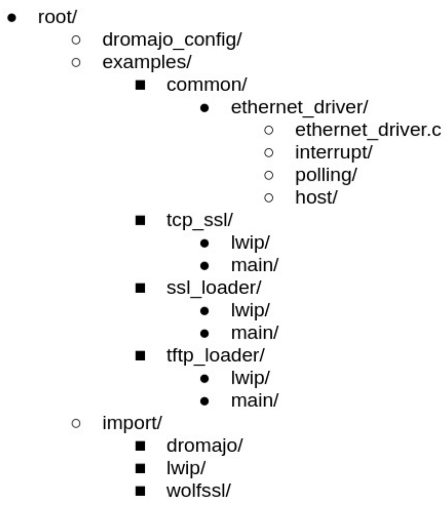


Figure 27: The Hierarchy an Ethernet Example. *ethernet_init*, *ethernet_recv* and *ethernet_send* are the three functions from the ethernet_driver API. They interface with two of the callbacks of the lwIP interface, which are input and output.



- root/
    - dromajo_config/
    - examples/
        - common/
            - ethernet_driver/
                - ethernet_driver.c
                - interrupt/
                - polling/
                - host/
        - tcp_ssl/
            - lwip/
            - main/
        - ssl_loader/
            - lwip/
            - main/
        - tftp_loader/
            - lwip/
            - main/
    - import/
        - dromajo/
        - lwip/
        - wolfssl/

Figure 28: The Root Directory of the Ethernet Examples

4.4.2 Ethernet Driver

Common Ethernet Driver

The below pseudo code comes from common/ethernet_driver/ethernet_driver.c.

As described in the porting section, lwIP uses *netif->input* and *netif->linkoutput* to communicate with an Ethernet driver. This file provides standard Ethernet device driver API for those two callbacks as well as the init function for netif.

There are three functions in this file, *ethernet_init*, *ethernet_recv* and *ethernet_send*. *ethernet_init* is called when lwIP initializes the Ethernet network interface. *ethernet_recv* checks if there is any incoming Ethernet packet. If there is one, it retrieves one packet and sends it into the lwIP network interface through *netif->input*. *ethernet_recv* should be called periodically so that lwIP can receive packets. *ethernet_send* will be assigned to *netif->linkoutput*, which will be called when lwIP wants to send out an Ethernet packet.

All these three functions call a function that starts with an underscore. These underscore functions are the actual Ethernet driver implementation that is provided by one of the three sub-folders: interrupt/, polling/ and host/.

```
// _ethernet_*: Actual Ethernet Driver Implementation
void _ethernet_init(struct netif *netif);
struct pbuf *_ethernet_recv(void);
int _ethernet_send(struct pbuf *pbuf);

// ethernet_*: Standard Ethernet Driver API
void ethernet_init(struct netif *netif)
{
    _ethernet_init(netif);
    netif_set_link_up(netif);
}
void ethernet_recv(struct netif *netif)
{
    struct pbuf *pbuf = _ethernet_recv();
    if(pbuf)
        netif->input(pbuf, netif);
}
err_t ethernet_send(struct netif *netif, struct pbuf *p)
{
    _ethernet_send(p);
}
```

Polling/

This implementation uses polling to detect and receive Ethernet packets. The receive flow is as follows: 1. Check if a packet has arrived 2. If yes, read out the packet size and allocate a pbuf. 3. Copy the packet into the pbuf. 4. Send an acknowledgement to the Ethernet controller. The transmit flow is as follows: 1. Write the packet and the size to the Ethernet controller. 2. Trigger a send to the controller.

```
void _ethernet_init(struct netif *netif)
{
    // empty
}

struct pbuf *_ethernet_recv(void)
{
    struct pbuf *pbuf = NULL;
    unsigned packet_size;
    // check if there is an arriving packet
    if(*(volatile unsigned *)eth_rx_pending_reg) {
        packet_size = *(volatile unsigned *)eth_rx_packet_size_reg;
        pbuf = pbuf_alloc(PBUF_RAW, packet_size, PBUF_RAM);
        // copy the packet from eth_rx_packet_reg into pbuf
        *(volatile unsigned *)eth_rx_pending_reg = 1; // ACK the packet
    }
    return pbuf;
}
int _ethernet_send(struct pbuf *pbuf)
{
    int send_size = pbuf->tot_len;
    // copy the packet from pbuf to eth_tx_packet_reg
    *(volatile unsigned *)eth_tx_size_reg = send_size;
    *(volatile unsigned *)eth_tx_send_reg = 1; // send out the packet
}
```

Interrupt/

This implementation uses interrupts to detect and receive Ethernet packets. The transmit flow is the same as the one in the polling implementation. The receive flow is as follows: 1. When a packet arrives, BlackParrot gets interrupted and the program enters the interrupt handler. 2. The program reads the PLIC claim register to find out the interrupt source. 3. If it is from the Ethernet controller, push the incoming packet to eth_fifo and send an acknowledgement to the controller. 4. Write the completion register to PLIC to complete this interrupt. 5. After leaving the interrupt handler, the program calls *ethernet_recv* in the mainloop. 6. The program pops the packet from eth_fifo in *ethernet_recv*.

Note that the interrupt handler does not pass the received packet directly into lwIP by calling *netif->input*, or allocate a piece of memory from *malloc* to hold the packet. Instead, it pushes the packet to eth_fifo first and then pops it back after leaving the handler. The reason behind it is that both *netif->input* and *malloc* cannot be called both in the mainloop context and in the interrupt context. For *malloc*, it is not reentrant according to the C standard, which means it cannot re-enter itself after being interrupted during some previous call to it. For *netif->input* (called in *ethernet_recv*), it can use the pbuf memory allocator internally, which can in turn be backed up by *malloc* depending on the lwIP configuration. Therefore both *netif->input* and *malloc* cannot be used in the interrupt handler, and a special FIFO, eth_fifo, is needed to solve the issue.

Figure 29 shows the overview of eth_fifo. eth_fifo consists of a read pointer, a write pointer and a static array of pbuf pointers. The array will be filled with valid pbuf pointers pointing to free pbuf structs during the program initialization. For the pop operation, it will return the pbuf struct in the slot pointed by the read pointer, allocate a new free pbuf from lwIP, and then update the read pointer. For the push operation, it will get a packet from the Ethernet controller and then store it into the free pbuf in the slot pointed by the write pointer. The pop operation will only be called from the mainloop, while the push operation will only be called from the interrupt handler. This way the packet allocation will not be called in more than one context, and hence preventing calls to non-reentrant functions.

```
// the Ethernet device is connecting to interrupt source #1 of PLIC
```

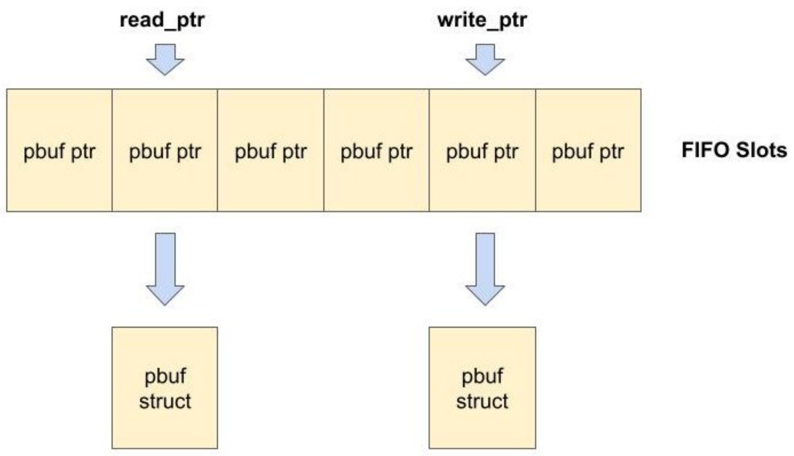


Figure 29: eth_fifo Overview

```
#define ETH_INT_ID 1
static void ethernet_interrupt_handler(void)
{
    // PLIC Claim operation
    int claim_id = *(volatile char *)plic_cc_reg;
    if(claim_id == ETH_INT_ID) {
        // check if there is an arriving packet
        if(*(volatile int *)eth_rx_pending_reg) {
            int rx_packet_size = *(volatile unsigned *)(eth_rx_packet_size_reg);
            eth_fifo_push((volatile char *)eth_rx_packet_reg, rx_packet_size);
            // Write 1 to ACK the received packet
            *(volatile int *)eth_rx_pending_reg = 1;
            // PLIC Completion operation
            *(volatile int *)plic_cc_reg = claim_id;
        }
    }
}
void _ethernet_init(struct netif *netif)
{
    // Register the S-mode interrupt handler, ethernet_interrupt_handler
    // Setting PLIC / Enabling S-mode interrupt
    // Init eth_fifo
}

struct pbuf *_ethernet_recv(void)
{
    return eth_fifo_pop();
}
int _ethernet_send(struct pbuf *pbuf)
{
    // same as polling/ethernet_driver_port.c
}
```

Host/

This implementation is for running on a Linux host. It uses the Linux raw socket API to send/receive Ethernet packets directly to/from an physical Ethernet interface on Linux.

```
int sockfd;
char buffer[2048];
struct sockaddr_ll socket_address;

void _ethernet_init(struct netif *netif)
{
    // Set up a Linux raw socket
}

struct pbuf *_ethernet_recv(void)
{
    struct pbuf *pbuf = NULL;
    // Non-blocking recv:
    int sz = recvfrom(sockfd, buffer, sizeof(buffer), MSG_DONTWAIT, NULL, NULL);
    if(sz > 0) {
        pbuf = pbuf_alloc(PBUF_RAW, sz, PBUF_RAM);
        memcpy(pbuf->payload, buffer, sz);
    }
    else {
        // No packet received
    }
    return pbuf;
}
int _ethernet_send(struct pbuf *pbuf)
{
    int send_size = pbuf->tot_len;
    // Get contiguous memory from pbuf
    void *buf = pbuf_get_contiguous(pbuf, buffer, sizeof(buffer), send_size, 0);
    return sendto(sockfd, buf, send_size, 0, (struct sockaddr *)&socket_address,
        sizeof(struct sockaddr_ll));
}
```

4.4.3 TCP and SSL Programs

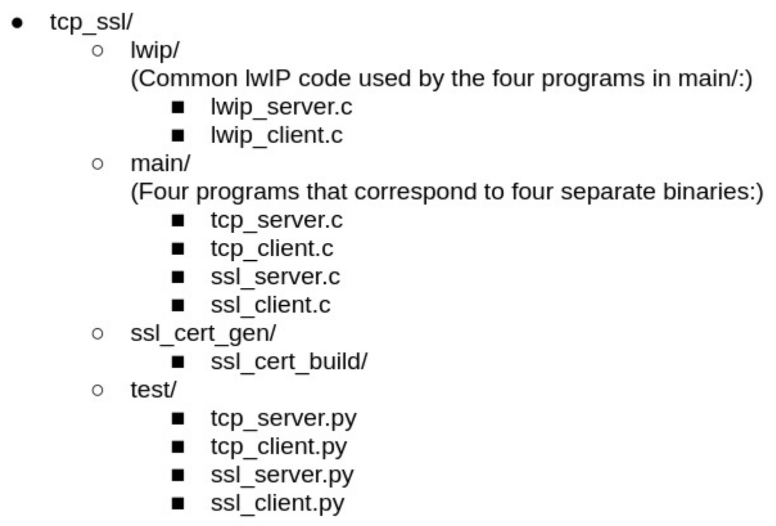
Overview

Figure 30 shows the directory structure under tcp_ssl/. This example produces four binaries, TCP server/client and SSL server/client. lwip_server.c is shared by both tcp_server.c and ssl_server.c, while lwip_client.c is shared by both tcp_client.c and ssl_client.c.

There are two standard interfaces between lwip_server/client and ssl_server/client. They are the server task API and the client task API. Both of them have exactly the same fields: *task_create* creates a task and returns the task ID; *task_scheduler* runs some runnable tasks one-by-one; *task_data_input* sends the TCP payload to the specific task; and *task_close* simply closes and frees the specific task.

Besides lwip/ and main/, there are also ssl_cert_gen/ and test/ in tcp_ssl/. ssl_cert_gen/ contains a Makefile that generates important SSL files for both the SSL server and client programs, including the public/private keys for both the server and CA, the CA certificate, the server certificate. After generating those files, it executes a generation program from wolfSSL to convert them into some C buffers (in my_certs_test.h) which will be loaded by the SSL server and client.

The sub-folder ssl_cert_build/ contains pre-built SSL files. test/ includes four simple python programs that are used for testing the functionality of all TCP and SSL programs.



- tcp_ssl/
  - lwip/
    (Common lwIP code used by the four programs in main/:)
    - lwip_server.c
    - lwip_client.c
  - main/
    (Four programs that correspond to four separate binaries:)
    - tcp_server.c
    - tcp_client.c
    - ssl_server.c
    - ssl_client.c
  - ssl_cert_gen/
    - ssl_cert_build/
  - test/
    - tcp_server.py
    - tcp_client.py
    - ssl_server.py
    - ssl_client.py

Figure 30: The Directory of the TCP/SSL Example

```
// Server Task API
struct tcp_server_task {
    int (*task_create) (struct tcp_pcb *newpcb);
    void (*task_scheduler)(void);
    int  (*task_data_input)(int task_id, struct pbuf *p);
    int  (*task_close)(int task_id);
};
// Client Task API
struct tcp_client_task {
    int (*task_create) (struct tcp_pcb *newpcb);
    void (*task_scheduler)(void);
    int  (*task_data_input)(int task_id, struct pbuf *p);
    int  (*task_close)(int task_id);
};
```

lwip_server:

The overview of the code is as follows: First, the server initializes the lwIP system and adds a network interface for the Ethernet hardware. During the call to *netif_add*, the init function *ethif_init* will be called to set some of the fields of the new network interface, including *netif->linkoutput*, which is set to one of the functions in the device driver API, *ethernet_send*. After the init function, all the packets forwarded to this interface will be sent out via *ethernet_send*.

After adding a network interface, the server brings the interface up and marks the start of an DHCP negotiation. Then it enters a loop that calls *PERIODIC_TASK* every cycle. *PERIODIC_TASK* consists of two function calls, *ethernet_recv* and *sys_check_timeouts*. The former is one of the functions in the device driver API, while the latter is the lwIP housekeeping function that handles all the time related operations. After entering the loop, the server keeps running until it gets an IP address from a DHCP server. Note that the entire DHCP negotiation process is completely hidden from the user, as *sys_check_timeouts* will call into lwIP and interact with the DHCP server through *ethernet_recv* and *netif->linkoutput* (which has been set to *ethernet_send*).

After getting an IP address, the server creates a TCP control block and binds it to the IP address. Then the server marks the TCP connection as LISTEN and sets the accept callback function to *tcp_accept_callback*, which will be called when an incoming connection is accepted.

Now all the configuration is set. The final step for the server is to enter an infinite loop that calls *PERIODIC_TASK* and the *task_scheduler* periodically. All the TCP connections are maintained in this loop and the user program will be notified through one of their registered callback functions if lwIP gets any event.

When a TCP connection is accepted, lwIP will inform the user by calling the accept callback function, *tcp_accept_callback*. This callback will spawn a task and associate the new task ID to the new TCP connection and set the receive callback, *tcp_recv_callback*, for the new TCP connection. If the argument p is not NULL, *tcp_recv_callback* passes the received TCP payload from that TCP connection to the user and then calls *tcp_recved* to inform its network peer that the user is ready to consume more TCP traffic. If the argument p is NULL, it indicates the connection has been closed.

tcp_recved is part of the TCP flow control mechanism. It grows the TCP receive window of 'pcb' by 'len' (figure 31) and is called when the program has taken 'len' bytes of data away from the receive buffer. In TCP, the receiver allocates a receive buffer for the received TCP data. In order not to overload the receiver, the sender gets the size of the receive window, which is essentially the free space left in the receive buffer, from the receiver, and limits the maximum counts of the sent-but-not-yet-ACKed data to the window size.

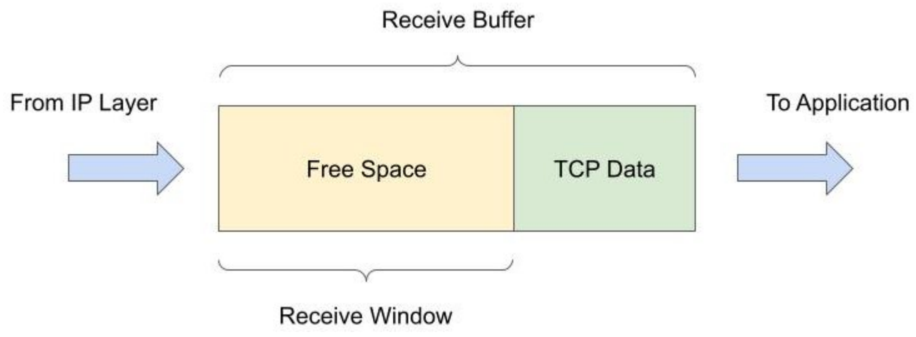


Figure 31: TCP Receive Window

Pseudo code of lwip_server

```
#define SERVER_PORT 12345
#define SERVER_MAC "\x00\x11\x22\x33\x44\x55"
#define PERIODIC_TASK(netif) do { \
    ethernet_recv((netif)); \
    sys_check_timeouts(); \
} while(0)
struct netif netif;
err_t tcp_recv_callback(void *arg, struct tcp_pcb *tpcb, struct pbuf *p,
        err_t err)
{
    int task_id = (int)arg;
    if(p) {
        tcp_server_task.task_data_input(task_id, p);
        tcp_recved(tpcb, p->tot_len);
        pbuf_free(p);
    } else {
        tcp_close(tpcb);
        tcp_server_task.task_close(task_id);
    }
    return ERR_OK;
}
err_t tcp_accept_callback(void *arg, struct tcp_pcb *tpcb, err_t err)
{
    int task_id = tcp_server_task.task_create(tpcb);
    tcp_recv(tpcb, tcp_recv_callback); // set callback
    tcp_arg(tpcb, (void *)task_id);
    return ERR_OK;
}
static err_t ethif_init(struct netif *netif)
{
    const char hwaddr[6] = SERVER_MAC;
    const int hwaddr_len = sizeof(hwaddr) / sizeof(*hwaddr);
    netif->state = NULL;
    netif->hwaddr_len = hwaddr_len;
    memcpy(netif->hwaddr, hwaddr, hwaddr_len);
    netif->mtu = 1500;
    memcpy(netif->name, "en", 2);
    netif->output = etharp_output;
    netif->linkoutput = ethernet_send;
    netif->flags |= (NETIF_FLAG_ETHERNET | NETIF_FLAG_ETHARP);
    ethernet_init(netif);
    return 0;
}
int main() {
    struct tcp_pcb *pcb = NULL;
    ip4_addr_t ipaddr;
    lwip_init();
    // netif->input is set to netif_input:
    netif_add(&netif, NULL, NULL, NULL, NULL, ethif_init, netif_input);
    netif_set_up(&netif);
    dhcp_start(&netif);
    while(dhcp_supplied_address(&netif) == 0) {
        PERIODIC_TASK(&netif);
    }
    pcb = tcp_new();
    ipaddr.addr = netif.ip_addr.addr;
    tcp_bind(pcb, &ipaddr, SERVER_PORT);
    pcb = tcp_listen(pcb);
    tcp_accept(pcb, tcp_accept_callback); // set callback
    // mainloop:
    while(1) {
        PERIODIC_TASK(&netif);
        tcp_server_task.task_scheduler();
    }
}
```

lwip_client:

lwip_client works pretty much in the same way as lwip_server, including using the same device driver API, task API, and having the same routine in the callbacks. The only main difference is that, in lwip_server, *tcp_bind*, *tcp_listen* and *tcp_accept* are used to set up the server, while in lwip_client, only *tcp_connect* is used for the client. Just like *tcp_accept*, *tcp_connect* sets up a callback that will be called when a TCP connection is made to the server.

Pseudo code of lwip_client

```
#define CLIENT_MAC "\x66\x55\x44\x33\x22\x11"
#define SERVER_IP "192.168.0.10"
#define SERVER_PORT 12345
#define PERIODIC_TASK(netif) do { \
    ethernet_recv((netif)); \
    sys_check_timeouts(); \
} while(0)

struct netif netif;

static err_t ethif_init(struct netif *netif)
{
    const char hwaddr[6] = CLIENT_MAC;
    const int hwaddr_len = sizeof(hwaddr) / sizeof(*hwaddr);
    netif->state = NULL;
    netif->hwaddr_len = hwaddr_len;
    memcpy(netif->hwaddr, hwaddr, hwaddr_len);
    netif->mtu = 1500;
    memcpy(netif->name, "en", 2);
    netif->output = etharp_output;
    netif->linkoutput = ethernet_send;
    netif->flags |= (NETIF_FLAG_ETHERNET | NETIF_FLAG_ETHARP);
    ethernet_init(netif);
    return 0;
}

err_t tcp_recv_callback(void *arg, struct tcp_pcb *tpcb, struct pbuf *p,
    err_t err)
{
    int task_id = (int)arg;
    if(p) {
        tcp_client_task.task_data_input(task_id, p);
        tcp_recved(tpcb, p->tot_len);
        pbuf_free(p);
    } else {
        tcp_close(tpcb);
        tcp_client_task.task_close(task_id);
    }
    return ERR_OK;
}

err_t tcp_connect_callback(void *arg, struct tcp_pcb *tpcb, err_t err)
{
    int task_id = tcp_client_task.task_create(tpcb);
    tcp_recv(tpcb, tcp_recv_callback); // set callback
    tcp_arg(tpcb, (void *)task_id);
    return ERR_OK;
}

int main () {
    lwip_init();
    // netif->input is set to netif_input:
    netif_add(&netif, NULL, NULL, NULL, NULL, ethif_init, netif_input);
    ip4_addr_t server_ipaddr = {ipaddr_addr(SERVER_IP)};
    netif_set_up(&netif);

    dhcp_start(&netif);
    while(dhcp_supplied_address(&netif) == 0) {
        PERIODIC_TASK(&netif);
    }
    pcb = tcp_new();
    // set callback
    tcp_connect(pcb, &server_ipaddr, SERVER_PORT, tcp_connect_callback);
    while(1) {
        PERIODIC_TASK(&netif);
        tcp_client_task.task_scheduler();
    }
}
```

tcp_server:

In tcp_server.c, each task struct has a task ID, a state variable for the state machine in *tcp_server_task_run*, and a private FIFO for received TCP payload. This file also defines the following functions, some of which are assigned to the server task API (see include/tcp_server_task.h):

- tcp_server_task_create (API: task_create)

  Create a new task struct which contains a new private FIFO.

- tcp_server_task_run_all (API: task_scheduler)

  Traverse the task struct list and run all the runnable tasks. All the tasks share the same routine, which is tcp_server_task_run.

- tcp_server_task_data_input (API: task_data_input)

  Push TCP payload to the private FIFO.

- tcp_server_task_close (API: task_close)

  Free the task struct.

- tcp_server_task_run

  This is the server routine for every TCP connection. The routine is a state machine and each task uses its state variable to resume its previous execution. The routine simply pops the TCP payload from the private FIFO and prints it out.

tcp_client:

tcp_client shares pretty much the same code logic as tcp_server.c except the prefix of each function name is tcp_client_task instead of tcp_server_task, and the function *tcp_client_task_run* is sending data instead of receiving. Before sending data through *tcp_write*, the client checks the remaining space in the send buffer by calling *tcp_sndbuf*.

ssl_server:

In ssl_server.c, each task struct has a task ID, a state variable for the state machine in *tcp_server_task_run*, and a private FIFO for received TCP payload. This file also defines the following functions, some of which are assigned to the server task API (see include/tcp_server_task.h):

- tcp_server_task_create (API: task_create)

  Create a new task struct which contains a new private FIFO. The function will also call *tcp_server_task_global_init* to select the TLS protocol and load the server certificate and the server private key from my_certs_test.h, which is located in ssl_cert_gen/ssl_cert_build/. The server certificate will be sent to and verified by the client. The underlying I/O callback functions for wolfSSL are also set here.

- tcp_server_task_run_all (API: task_scheduler)

  Traverse the task struct list and run all the runnable tasks. All the tasks share the same routine, which is *tcp_server_task_run*.

- tcp_server_task_data_input (API: task_data_input)

  Push TCP payload to the private FIFO. The payload will be popped out later by *ssl_media_recv*.

- tcp_server_task_close (API: task_close)

  Free the task struct and the SSL object.

- tcp_server_task_run

  This is the server routine for every SSL connection. The routine is a simple state machine and each task maintains its own state with its state variable. The first step for the server is to make an SSL connection with *wolfSSL_accept*. After that, the server will forward its state and keep reading the data from the client with *wolfSSL_read*. Since the underlying I/O callbacks (*ssl_media_send*, *ssl_media_recv*) are non-blocking, both *wolfSSL_accept* and *wolfSSL_read* can return an error indicating that the operation has not yet finished. Therefore, those operations have to be called repeatedly until they succeed (or fail with any other error). Note that *tcp_server_task_run* does not contain any loop to repeatedly call those non-blocking functions. *tcp_server_task_run* will be called repeatedly in the mainloop in lwip_server.

- ssl_media_send

  This is the actual send function that wolfSSL would call when it wants to send out data to the underlying transport layer. This function performs non-blocking send. It will return an error indicating the operation would block if the data cannot be sent.

- ssl_media_recv

  This is the actual recv function that wolfSSL would call when it wants to receive data from the underlying transport layer. This function pops the received TCP payload from the private FIFO specific to a SSL connection and then sends it to wolfSSL. This function performs non-blocking recv. It will return an error indicating the operation would block if there is no data available from the FIFO.

Pseudo code of ssl_server

```
#include "my_certs_test.h"

WOLFSSL_CTX *ctx = NULL;
int ssl_media_send(WOLFSSL *ssl, char *buf, int sz, void *ctx);
int ssl_media_recv(WOLFSSL *ssl, char *buf, int sz, void *ctx);
void tcp_server_task_global_init(void)
{
    wolfSSL_Init();
    WOLFSSL_METHOD *method = wolfTLSv1_2_server_method();
    ctx    = wolfSSL_CTX_new(method);
    // set I/O callbacks
    wolfSSL_CTX_SetIORecv(ctx, ssl_media_recv);
    wolfSSL_CTX_SetIOSend(ctx, ssl_media_send);
    // load the server certificate
    wolfSSL_CTX_use_certificate_buffer(ctx, server_cert_pem,
        sizeof(server_cert_pem), WOLFSSL_FILETYPE_PEM);
    // load the server private key
    wolfSSL_CTX_use_PrivateKey_buffer(ctx, server_private_key_pem,
        sizeof(server_private_key_pem), WOLFSSL_FILETYPE_PEM);
}

int tcp_server_task_create(struct tcp_pcb *newpcb)
{
    static int tcp_server_task_global_inited = 0;
    int task_id;
    if(tcp_server_task_global_inited == 0) {
        tcp_server_task_global_init();
        tcp_server_task_global_inited = 1;
    }
    // assign a free ID to task_id
    // init the new task struct
    // set task initial state to SSL_PROGRESS_ACCEPT
}
void tcp_server_task_run(int task_id)
{
    WOLFSSL *ssl;
    int *progress
    int ret, read_count;
    // get ssl and progress with task_id
    switch (*progress) {
        case SSL_PROGRESS_ACCEPT:
            ret = wolfSSL_accept(ssl);
            ret = wolfSSL_get_error(ssl, ret);
            if(ret == WOLFSSL_ERROR_NONE)
                *progress = SSL_PROGRESS_IO;
            else if(ret != WOLFSSL_ERROR_WANT_READ &&
                        ret != WOLFSSL_ERROR_WANT_WRITE)
                *progress = SSL_PROGRESS_ERROR;
            break;
        case SSL_PROGRESS_IO:
            read_count = wolfSSL_read(ssl, mesg, sizeof(mesg) - 1);
            ret = wolfSSL_get_error(ssl, read_count);
            if(ret == WOLFSSL_ERROR_NONE) {
                // mesg received
            }
            else if(ret != WOLFSSL_ERROR_WANT_READ &&
                        ret != WOLFSSL_ERROR_WANT_WRITE)
                *progress = SSL_PROGRESS_ERROR;
            break;
        case SSL_PROGRESS_ERROR:
            break;
    }
}
int ssl_media_send(WOLFSSL *ssl, char *buf, int sz, void *ctx)
{
    struct tcp_pcb *tpcb;
    int free, len, ret;
    int task_id = wolfSSL_get_fd(ssl);
    // get tpcb with task_id
    free = tcp_sndbuf(tpcb);
    len = (sz < free) ? sz : free;
    ret = tcp_write(tpcb, buf, len, TCP_WRITE_FLAG_COPY);
    if(ret != ERR_OK) {
        // would block
        return WOLFSSL_CBIO_ERR_WANT_WRITE;
    }
    return len;
}
int ssl_media_recv(WOLFSSL *ssl, char *buf, int sz, void *ctx)
{
    int task_id = wolfSSL_get_fd(ssl);
    int count;
    // get the private FIFO associated with task_id
    // pop the data from the FIFO and get the count of the data
    if(count == 0) {
        // would block
        return WOLFSSL_CBIO_ERR_WANT_READ;
    }
    return count;
}
```

ssl_client

ssl_client shares pretty much the same code logic as ssl_server.c. The main difference is that the client loads the CA certificate from my_certs_test.h during its initialization (in *tcp_client_task_global_init*). The CA certificate contains the CA's public key and it is used to verify if the server's certificate really comes from the CA. The other difference is that the client uses *wolfSSL_connect* to connect to the server.

Pseudo code of ssl_client

```
#include "my_certs_test.h"

WOLFSSL_CTX *ctx = NULL;
```

```
char mesg[1024] = ...;
int mesg_size = ...;
int mesg_idx = 0;

void tcp_client_task_global_init(void)
{
    wolfSSL_Init();
    WOLFSSL_METHOD *method = wolfTLSv1_2_client_method();
    ctx    = wolfSSL_CTX_new(method);
    wolfSSL_CTX_SetIORecv(ctx, ssl_media_recv);
    wolfSSL_CTX_SetIOSend(ctx, ssl_media_send);
    // load the CA certificate
    wolfSSL_CTX_load_verify_buffer(ctx, ca_cert_pem, sizeof(ca_cert_pem),
    WOLFSSL_FILETYPE_PEM);
}
void tcp_client_task_run(int task_id)
{
    WOLFSSL *ssl;
    int *progress;
    // get ssl and progress with task_id
    switch (*progress) {
        case SSL_PROGRESS_CONNECT:
            ret = wolfSSL_connect(ssl);
            // if succeeds, go to SSL_PROGRESS_IO
            // if error happens, go to SSL_PROGRESS_ERROR
        case SSL_PROGRESS_IO:
            // wolfSSL_write(...)
            // if error happens, go to SSL_PROGRESS_ERROR
        case SSL_PROGRESS_ERROR:
            break;
    }
}
```

4.4.4 Boot Loaders

The second and third examples of lwIP and wolfSSL are the two boot loaders for the BlackParrot System on Module. They are the first-stage boot loader in tftp_loader and the second-stage boot loader in ssl_loader. The introduction of the BlackParrot System on Module will be given first. Then the two boot loaders will be explained later.

BlackParrot System on Module

The BlackParrot System on Module (BP SOM) consists of only a BlackParrot core, an Ethernet controller, a piece of tiny ROM and RAM. This lightweight design leads to low cost and little power consumption and is suitable for embedded applications. However, if users would like to run a Linux distribution on a BP SOM, the software booting process would be challenging as there would be no hard drive storing the Linux kernel and all the other system data. What makes things even more difficult is that the tiny ROM in the BP SOM should be made as small as possible, possibly no larger than 64 KiB, which is too small for holding a regular boot loader. In order to address the above issues, two boot loaders along with the Linux Network File System (NFS) have been introduced to the booting process.

Figure 32 shows the booting process of a Linux distro on the BP SOM. The middle row shows the current program running in the BP SOM at a specific stage, while the bottom row shows the three different servers. Since the first-stage boot loader is stored in ROM, the booting process starts by relocating the first-stage boot loader from ROM to RAM. Then the first-stage boot loader downloads the second-stage boot loader through a protocol called Trivial File Transfer Protocol (TFTP). TFTP is a client server protocol: the client sends read or write requests to the server that hosts the files. TFTP is also a lightweight protocol running on UDP. It has been frequently used in network booting as it can be implemented with a very small memory footprint.

After downloading the second-stage boot loader, the second-stage boot loader downloads the Linux kernel from the SSL server and executes it. At the end of execution, the Linux kernel does the NFS mounting: it mounts the root directory of the Linux distro from the NFS server as its root. Now every access to the root file system happens on the network, as if the root file system were stored on a local hard drive. Finally, the kernel runs an init script in the root to boot the Linux Distro.

The purpose of passing the Linux kernel through an SSL connection is to ensure the kernel image is not corrupted by any malicious attack. The reason why the SSL client is not used as the first-stage boot loader is that the code size for both the SSL and TCP (SSL typically runs on TCP) layers is too large to fit into a 64-KiB ROM. Therefore a more lightweight protocol, TFTP, is chosen for the first stage instead. In order to make sure that the second-stage loader passed by the TFTP server will not be corrupted, a SHA256 checksum of the loader is written in ROM beforehand with the TFTP client program. This allows the TFTP client program to discover any errors by comparing the checksum of the received loader against the pre-calculated checksum.

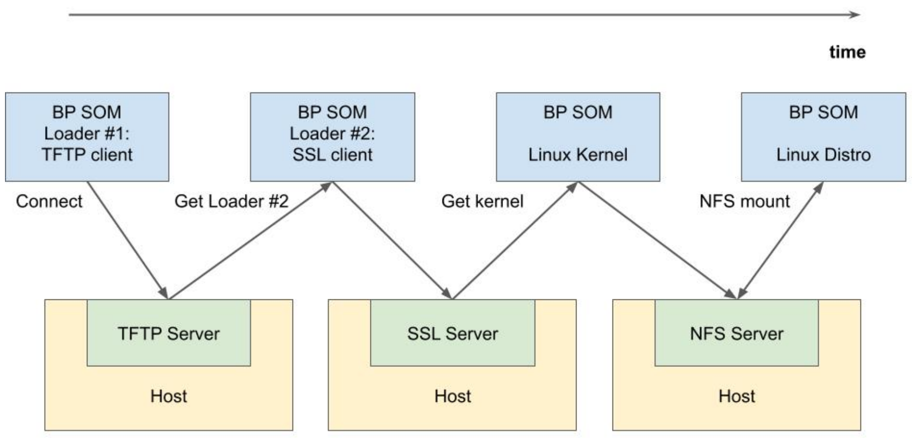


Figure 32: The Booting Procces of a Linux Distro on the BlackParrot System on Module

The TFTP Loader

lwip/lwip_client.c works in the similar way as the one in tcp_ssl/, except that the lwip_client.c for the TFTP loader uses the UDP API instead of the TCP API. The first-stage boot loader consists of two parts: the TFTP client that downloads the second-stage boot loader, and the program loader that relocates the second-stage boot loader to wherever it should be. Figure 33 shows the TFTP state machine in the first-stage boot loader. The client always waits for the receiving of the DATA packet (which contains the second-stage boot loader) before it sends out an ACK to the server, and the server also waits for the receiving of the ACK before it sends out the next DATA packet. This ensures that the received data will not be reordered. After receiving all the DATA packets, the boot loader verifies the SHA256 checksum of the received boot loader and starts loading it.

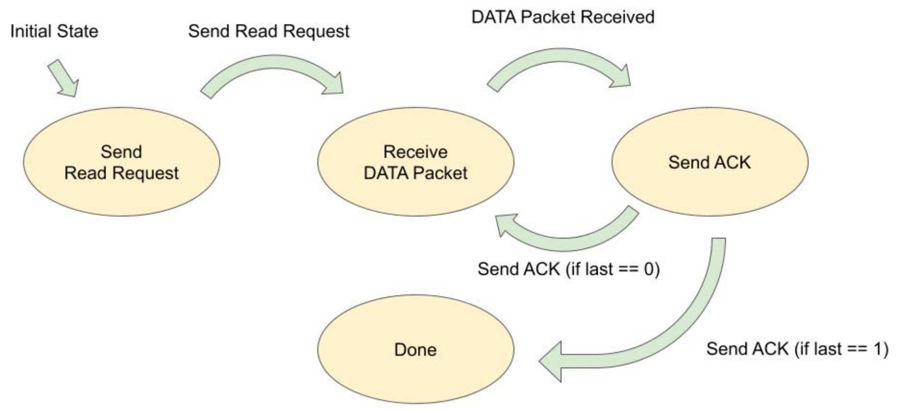


Figure 33: The FSM within the TFTP Loader

In order to minimize the code size of the TFTP loader as much as possible, the following four steps have been taken for the TFTP loader:

1. Eliminate the use of BlackParrot Newlib.

2. Remove unnecessary features, for example, TCP protocol and IP fragmentation, by specifying options in lwipopts.h. IP fragmentation is unnecessary because the TFTP blocksize used in the loader is smaller than the maximum size a UDP packet can have without fragmentation.

3. Remove the code of the internal lwIP heap memory allocator by reusing the existing lwIP memory pool allocator. As described in the previous section, there are several implementations for the lwIP heap memory allocator. One of the implementations is to use the lwIP memory pool allocator for the lwIP heap memory allocation. This can be done by setting MEM_USE_POOLS to 1 in lwipopts.h, and defining a

configuration file lwippools.h that specifies the schema of the memory pools. For example, define two memory pools, one with 50 blocks, 64 bytes for each, and the other with 25 blocks, 88 bytes for each. By reusing the code, the total code size is further reduced.

4. Implement only necessary features of the TFTP protocol. In our case, support only the read requests, as the TFTP loader is only required to read a file from the server.

The SSL Loader

Similar to the first-stage boot loader, the second-stage loader also consists of two parts: the SSL client that loads the Linux binary and the program loader that loads the Linux binary to where it should be. The SSL client part of the boot loader uses the SSL client in the tcp_ssl example as the template. Therefore the description is omitted here.

5 Validation and Results

The combination of BlackParrot, BlackParrot Ethernet Controller and the RISC-V PLIC module has been verified both in software simulation and on Zedboard, under the ZynqParrot framework. Figure 34 shows the final block diagram with all the modules integrated. The ETH block is the BlackParrot Ethernet controller with an AXI client (slave) adapter, whereas the PLIC block is the RISC-V PLIC module from OpenTitan with an AXI master adapter and an AXI client adapter. Although not shown in the figure, the RGMII interface of the Ethernet controller is connected to the FMC interface on Zedboard, and the TX and RX IRQ line of the Ethernet controller are ORed together before connecting to the interrupt source 1 of the PLIC module. The clock frequencies for the BlackParrot, the Ethernet Controller (except the RGMII part of the circuits), the PLIC module and the AXI connections are all 20 MHZ.

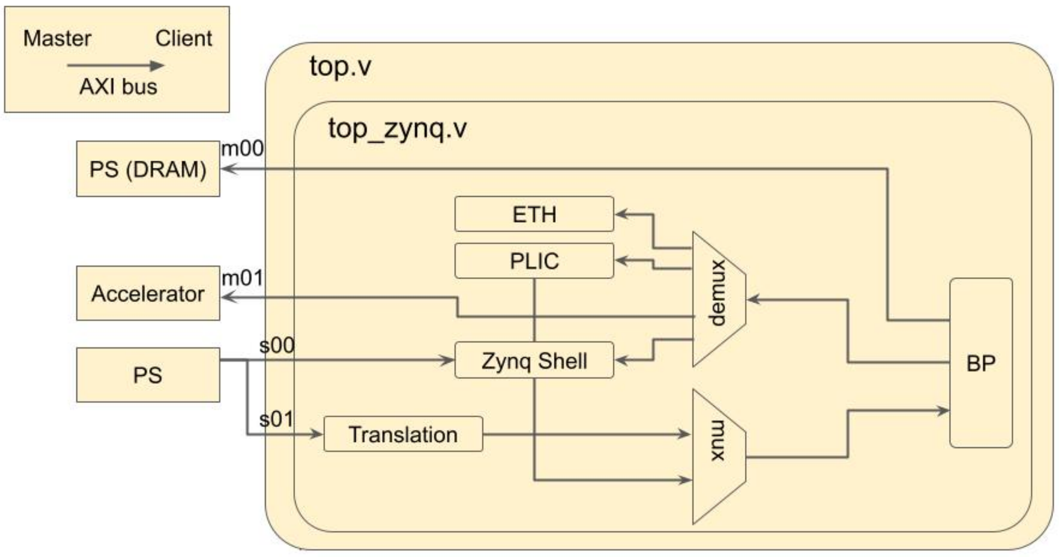


Figure 34: ZynqParrot with Ethernet Controller and PLIC

Resource	Utilization	Available	Utilization %
LUT	38921	53200	73.16
LUTRAM	3386	17400	19.46
FF	20207	106400	18.99
BRAM	123.5	140	88.21

Table 1: ZynqParrot Utilization

Resource	Utilization	ZynqParrot %
LUT	849	2.18
LUTRAM	0	0.00
FF	981	4.85
BRAM	9.5	7.69

Table 2: The BlacParrot Ethernet Utilization

5.1 FPGA Synthesis Reports

5.1.1 Zedboard Utilization

Table 1 shows the total utilization of BlackParrot, Ethernet controller and PLIC module and other modules under ZynqParrot. Table 2 and table 3 show the relative utilization compared to the total of ZynqParrot.

5.2 Quality of the RX RGMII signals

Even if the BlackParrot Ethernet controller supports 1G Ethernet, without having good quality of the RGMII signals, the speed would be severely degraded. The Ethernet controller does Cyclic Redundancy Check (CRC) whenever it receives a packet. If there is data corruption in the packet, possibly due to some noises in the Ethernet cable or improper latch of some RX RGMII signals, the hardware will simply discard the packet and

Resource	Utilization	ZynqParrot %
LUT	3	0.01
LUTRAM	0	0.00
FF	16	0.08
BRAM	0	0.00

Table 3: The RISC-V PLIC Module Utilization

result in a packet loss.

The original MAC module from Alex already provides signals that would indicate whether such a packet loss is happening on the RX side. Those signals have been used during the software tests (which will be described in the next section) to evaluate the quality of the RX RGMII signals. The end results showed that there was no such packet loss on the RX side. For the quality of TX RGMII signals, it is hard to show if there is data corruption happening there, since there is no feedback signal from the receiver that would indicate the data has been corrupted.

5.3 Software Tests

- Linux Kernel 5.15 (with Liteeth driver and the RISC-V PLIC driver)
  After booting up the Linux kernel on Zedboard, the Ethernet connection was tested by running a TCP server written in C that sends 4 MiB of data to a host. The Ethernet connection was also tested by doing a Linux NFS boot. First, an NFS server that would provide the root directory of a Linux distro from Yocto was launched on a host computer. Then the Linux kernel began to boot on Zedboard.

- The TCP/SSL Examples
  During the test for TCP/SSL server, multiple TCP/SSL client programs written in Python connected to the server on Zedboard one by one and then they sent/received messages to/from the server at the same time. The clients also connected and disconnected multiple times to ensure the server could handle the disconnection properly.
  During the test for the TCP/SSL client, a TCP/SSL server written in Python launched on a host first and then the TCP/SSL client on Zedboard sent data to the server.

- The BlackParrot System on Module
  The complete boot flow was tested on Zedboard: The first-stage boot loader loaded the second-stage boot loader from a busybox TFTP server running on a host and verified its SHA256 checksum. The second-stage boot loader then ran and loaded the Linux binary from a python SSL server running on a host. After booting up the

Linux kernel, the kernel started NFS booting from an NFS server on a host to boot the Linux Distro from Yocto.

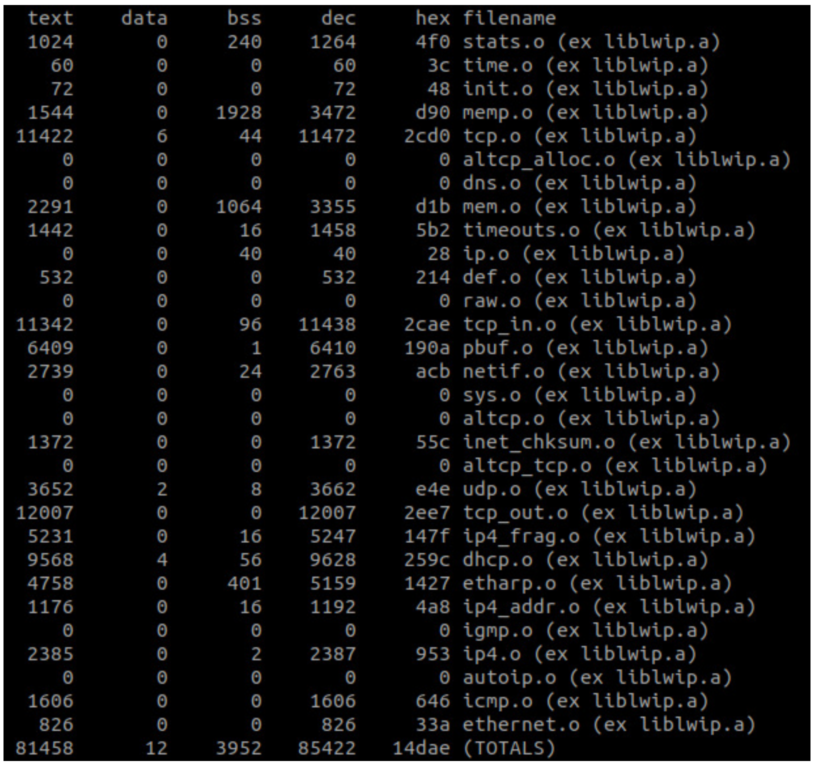
5.4 Software Footprint

5.4.1 lwIP Library Footprint

Figure 35 shows the breakdown of the lwIP library footprint. The data in the figure was generated by the following command 'size -t'. Each column, from left to right, shows the text section, the data section, the bss section and the total size of the previous three sections in decimal and hexadecimal. Each row shows the size of each component in the lwIP library, and the total sizes can be seen at the bottom row. The library has been compiled by "gcc" with "-Os" and without RISC-V compressed instruction set (as BlackParrot currently does not support the RISC-V C Extension). It has also been compiled with "-fno-common", so that the uninitialized data (for example, the static memory managed by the two lwIP memory allocators) is put in the bss section.

This library has been tuned for the TCP/SSL programs from my first example (and assuming that the server program can only make up to 4 TCP connections at the same time). The total text size of the library is roughly 79.5KiB, and the total size of data and bss is roughly 3.9KiB. The two main sources for the bss section are the static memory from the lwIP heap allocator (mem.o) and the static memory from the lwIP memory pool allocator (memp.o). The MEM_SIZE macro that specifies the maximum size of the lwIP heap, and the MEMP_NUM_* macros that specify the maximum numbers of elements in each of the lwIP memory pool, have been specified in lwipopts.h to minimize the sizes of the two bss sections. Those memory limits should be set properly based on the actual applications. They can be found by setting MEM_STATS to 1 in lwipopts.h and dumping the memory stats with *stats_display*.

In our specific setting, the dynamic memory footprint increases 216 bytes per TCP connection, as each TCP connection is represented by a TCP control block, and the control block is 216 bytes.



```
   text    data     bss     dec     hex filename
   1024       0     240    1264     4f0 stats.o (ex liblwip.a)
     60       0       0      60      3c time.o (ex liblwip.a)
     72       0       0      72      48 init.o (ex liblwip.a)
   1544       0    1928    3472     d90 memp.o (ex liblwip.a)
  11422       6      44   11472    2cd0 tcp.o (ex liblwip.a)
      0       0       0       0       0 altcp_alloc.o (ex liblwip.a)
      0       0       0       0       0 dns.o (ex liblwip.a)
   2291       0    1064    3355     d1b mem.o (ex liblwip.a)
   1442       0      16    1458     5b2 timeouts.o (ex liblwip.a)
      0       0      40      40      28 ip.o (ex liblwip.a)
    532       0       0     532     214 def.o (ex liblwip.a)
      0       0       0       0       0 raw.o (ex liblwip.a)
  11342       0      96   11438    2cae tcp_in.o (ex liblwip.a)
   6409       0       1    6410    190a pbuf.o (ex liblwip.a)
   2739       0      24    2763     acb netif.o (ex liblwip.a)
      0       0       0       0       0 sys.o (ex liblwip.a)
      0       0       0       0       0 altcp.o (ex liblwip.a)
   1372       0       0    1372     55c inet_chksum.o (ex liblwip.a)
      0       0       0       0       0 altcp_tcp.o (ex liblwip.a)
   3652       2       8    3662     e4e udp.o (ex liblwip.a)
  12007       0       0   12007    2ee7 tcp_out.o (ex liblwip.a)
   5231       0      16    5247    147f ip4_frag.o (ex liblwip.a)
   9568       4      56    9628    259c dhcp.o (ex liblwip.a)
   4758       0     401    5159    1427 etharp.o (ex liblwip.a)
   1176       0      16    1192     4a8 ip4_addr.o (ex liblwip.a)
      0       0       0       0       0 igmp.o (ex liblwip.a)
   2385       0       2    2387     953 ip4.o (ex liblwip.a)
      0       0       0       0       0 autoip.o (ex liblwip.a)
   1606       0       0    1606     646 icmp.o (ex liblwip.a)
    826       0       0     826     33a ethernet.o (ex liblwip.a)
  81458      12    3952   85422   14dae (TOTALS)
```

Figure 35: lwIP Library Footprint

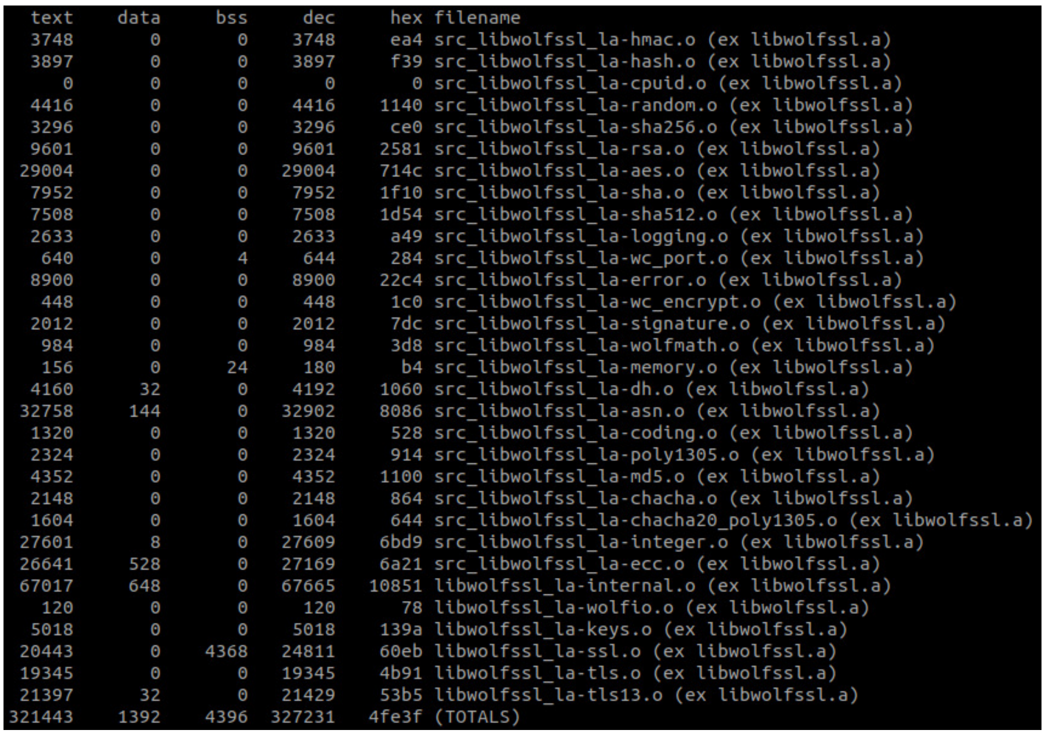
5.4.2 wolfSSL Library Footprint

Figure 36 shows the breakdown of the wolfSSL library footprint. Similar to the lwIP library, the data from the figure was generated by the following command 'size -t'. The library has been compiled by "gcc" with "-Os" and without RISC-V compressed instruction set. It has also been compiled with "-fno-common". The total text size of the library is roughly 313.9 KiB, and the total size of data and bss is roughly 5.7 KiB.

With one SSL connection, the dynamic memory footprint is roughly 14.8 KiB. With more than one SSL connection, the dynamic memory footprint increases 3.1 KiB/s per SSL connection.

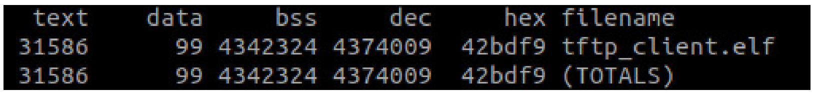
5.4.3 The TFTP Loader Footprint

Figure 37 shows the footprint of the TFTP loader. The size of the text section is roughly 30.8 KiB (this amount of code will be put in the ROM). The bss section is roughly 4.1MiB. 4MiB is allocated for storing the second-stage boot loader.



```
  text    data     bss     dec     hex filename
  3748       0       0    3748     ea4 src_libwolfssl_la-hmac.o (ex libwolfssl.a)
  3897       0       0    3897     f39 src_libwolfssl_la-hash.o (ex libwolfssl.a)
     0       0       0       0       0 src_libwolfssl_la-cpuid.o (ex libwolfssl.a)
  4416       0       0    4416    1140 src_libwolfssl_la-random.o (ex libwolfssl.a)
  3296       0       0    3296     ce0 src_libwolfssl_la-sha256.o (ex libwolfssl.a)
  9601       0       0    9601    2581 src_libwolfssl_la-rsa.o (ex libwolfssl.a)
 29004       0       0   29004    714c src_libwolfssl_la-aes.o (ex libwolfssl.a)
  7952       0       0    7952    1f10 src_libwolfssl_la-sha.o (ex libwolfssl.a)
  7508       0       0    7508    1d54 src_libwolfssl_la-sha512.o (ex libwolfssl.a)
  2633       0       0    2633     a49 src_libwolfssl_la-logging.o (ex libwolfssl.a)
   640       0       4     644     284 src_libwolfssl_la-wc_port.o (ex libwolfssl.a)
  8900       0       0    8900    22c4 src_libwolfssl_la-error.o (ex libwolfssl.a)
   448       0       0     448     1c0 src_libwolfssl_la-wc_encrypt.o (ex libwolfssl.a)
  2012       0       0    2012     7dc src_libwolfssl_la-signature.o (ex libwolfssl.a)
   984       0       0     984     3d8 src_libwolfssl_la-wolfmath.o (ex libwolfssl.a)
   156       0      24     180      b4 src_libwolfssl_la-memory.o (ex libwolfssl.a)
  4160      32       0    4192    1060 src_libwolfssl_la-dh.o (ex libwolfssl.a)
 32758     144       0   32902    8086 src_libwolfssl_la-asn.o (ex libwolfssl.a)
  1320       0       0    1320     528 src_libwolfssl_la-coding.o (ex libwolfssl.a)
  2324       0       0    2324     914 src_libwolfssl_la-poly1305.o (ex libwolfssl.a)
  4352       0       0    4352    1100 src_libwolfssl_la-md5.o (ex libwolfssl.a)
  2148       0       0    2148     864 src_libwolfssl_la-chacha.o (ex libwolfssl.a)
  1604       0       0    1604     644 src_libwolfssl_la-chacha20_poly1305.o (ex libwolfssl.a)
 27601       8       0   27609    6bd9 src_libwolfssl_la-integer.o (ex libwolfssl.a)
 26641     528       0   27169    6a21 src_libwolfssl_la-ecc.o (ex libwolfssl.a)
 67017     648       0   67665   10851 libwolfssl_la-internal.o (ex libwolfssl.a)
   120       0       0     120      78 libwolfssl_la-wolfio.o (ex libwolfssl.a)
  5018       0       0    5018    139a libwolfssl_la-keys.o (ex libwolfssl.a)
 20443       0    4368   24811    60eb libwolfssl_la-ssl.o (ex libwolfssl.a)
 19345       0       0   19345    4b91 libwolfssl_la-tls.o (ex libwolfssl.a)
 21397      32       0   21429    53b5 libwolfssl_la-tls13.o (ex libwolfssl.a)
321443    1392    4396  327231   4fe3f (TOTALS)
```

Figure 36: wolfSSL Library Footprint



```
  text    data     bss     dec     hex filename
 31586      99 4342324 4374009  42bdf9 tftp_client.elf
 31586      99 4342324 4374009  42bdf9 (TOTALS)
```

Figure 37: The TFTP Loader Footprint

5.5 Network Throughput

The network throughputs were measured on Zedboard with BlackParrot running at 20 MHZ. The Ethernet speed ran at 1Gbps and was connected to a 1G switch. There was also a Linux host computer connected to the switch.

- lwIP UDP Throughput

  With continuous calls to udp_send with 1472 bytes of data (This is the maximum size one can get to send out UDP packets without fragmentation) and PERIODIC_TASK, the TX UDP throughput is roughly 1707 KiB/s.

  When the host sends UDP packets with size 1472 bytes continuously, the RX UDP throughput is roughly 1791 KiB/s.

- lwIP TCP Throughput

When transmitting files between the host, the RX TCP throughput is roughly 918.0 KiB/s, while the TX TCP throughput is roughly 1024 KiB/s.

- wolfSSL Throughput

  When transmitting files between the host, the RX SSL throughput is roughly 78.3 KiB/s, while the TX SSL throughput is roughly 50.8 KiB/s.

- Linux Throughput

  When sending data to the host through the Linux TCP socket, the TX throughput is roughly 331.1KiB/s.

- Linux NFS Throughput

  When running "dd if=/dev/zero of=test_file bs=X count=Y", where, X * Y equals to 2MiB (that is, the total size is fixed to 2MiB), the throughput are as follows:

  - X=512: 59.9 KiB/s
  - X=1024: 74.7 KiB/s
  - X=2048: 82.4 KiB/s
  - X=4096: 86 KiB/s
  - X=8192: 90 KiB/s
  - X=16384: 97.6 KiB/s
  - X=32768: 95.4 KiB/s

The ideal throughput of 1G Ethernet should be 125 MB/s (1 Gbps / 8 bit). However, since ZynqParrot is running at 20 MHZ and the AXI bus to the Ethernet controller is 32-bit, the ideal throughput under this hardware setting can only be 80 MB. Other factors could also further reduce the throughput: The first one is the DEMUX module. The module currently does not support multiple outstanding AXI requests, that is, it can only handle requests one by one. Therefore, the cycles needed to handle one single request is equal to the roundtrip latency between the DEMUX and the Ethernet controller, which is 4 cycles. Therefore the throughput reduces from 80 MB to 20 MB. The second one is the software driver: The above analysis assumes that the CPU can write data with full

32-bit length to the controller one after another. However, the software driver cannot always send the data back-to-back since it also has to move the data from other memory locations and check the sending progress in order not to overflow the write buffer. Last but not least, the CPU has to spend time processing Ethernet packets. These would all affect the actual Ethernet throughput.

6 Conclusion and Future Works

This thesis work brings the complete hardware and software solution of making Ethernet connections to BlackParrot. For hardware, an Ethernet controller based on the MAC modules from Alex has been created. The RISC-V PLIC module from OpenTitan has also been tested with BlackParrot. Both hardware modules are open-source and can be targeted at both ASIC and FPGA. Both hardware modules have also been tested on an FPGA and were able to run various kinds of Ethernet software described below. For software, the TCP/IP library lwIP and SSL library wolfSSL have been ported to BlackParrot, both of which are open-source, and several Ethernet examples using those libraries have been created. Those examples are able to make connections to real-world programs like TCP/SSL Python programs and the TFTP server from busybox. Besides these two libraries, the Linux kernel 5.15 along with the Linux Liteeth Ethernet driver and the Linux RISC-V PLIC driver have also been tested with the hardware. The kernel is able to NFS boot a Linux distro from an NFS server, without the need to have any BlackParrot-specific changes to Linux.

Although the entire network stack has been brought to BlackParrot, there are plenty of things that can be improved:

1. Currently, the hardware modules have only been tested on FPGA. In the future, they should also be tested in ASIC.

2. The BlackParrot Ethernet controller does not support DMA. Implementing one could improve the network throughput a lot. However, adding DMA support might indicate custom changes are needed for the Ethernet driver in Linux.

3. The infrastructure of the Ethernet ZynqParrot needs to be changed. The original version is shown in figure 34. In order to have a more flexible design, the Ethernet controller and the PLIC device can be moved outside the top module and be connected through the Xilinx AXI SmartConnect IPs, as shown in figure 38. With this design, all the connections of future peripheral devices can be done without changing anything within the top module. However, there is one downside to this design:

the simulation framework will no longer fork for the Ethernet controller and PLIC module, as ZynqParrot can only do simulation for modules within the top module. Therefore, both the hardware infrastructure and the simulation framework need adjustment.

4. The code size of the first-stage boot loader of the BlackParrot SOM can possibly be reduced further with compression. The idea is to store the compressed boot loader into ROM, and decompress it before running the boot loader. This idea needs to be evaluated carefully as the extra code for the decompression might outweigh the benefit.

5. The throughput of the AXI DEMUX and MUX can be further improved by allowing multiple outstanding AXI requests instead of handling one at a time. Other than the clock speed, this is currently the bottleneck of the Ethernet connection speed.

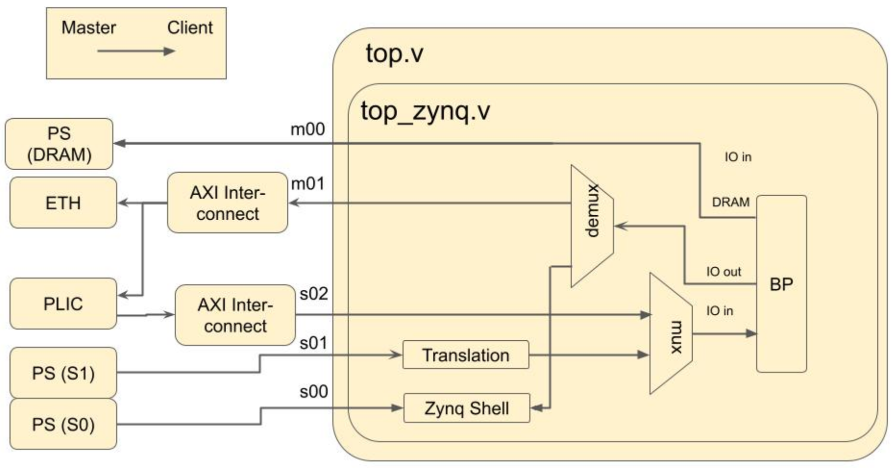


Figure 38: New ZynqParrot Architecture

7 Acknowledgement

First of all, I would like to thank my parents, Chung-Chin Chiang and Wen-Yuh Chueh and my brother Yuan-Hui Chueh for their tremendous support. The pandemic time has been really frustrating to me, and without their mental support, I will not be able to stick to my graduate study.

I would like to thank my advisor, Prof. Michael Taylor for his guidance and support. Due to the pandemic, Michael and I had discussion online and did not meet in-person much. However, I was still able to find him easily online whenever I needed help from him. I also really like the way Michael gives advice to the Master students: he would only point out the pig picture of a task, and then he will let the student figure out the rest on their own. This really helped my growth as an independent thinker.

Also many thanks to Daniel Petrisko for being my main mentor throughout my graduate study. He has been really patient and is always willing to spend time explaining even the most basic things to me to help me overcome the steep learning curve of doing hardware research. Without his support, I would not have a smooth learning experience in Bespoke Silicon Group.

Lastly, I would like to thank my friends, Yi-Wei and Chang-Xian, and all my other family members, especially Winnie and John, for their constant mental support. Whenever I felt frustrated about my graduate school life and needed them the most, they would always show up and support me going through the tough time.

Portions of this work were partially supported by Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under agreement numbers FA8650-18-2-7863 and FA8650-18-2-7856; NSF grants SaTC-1563767, SaTC-1565446, and the DARPA/SRC JUMP ADA Center. This work intersects and leverages research and infrastructure created by the members of the Bespoke Silicon Group, spanning across accelerators ([27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44]), ASIC Clouds ([45, 46, 47, 48, 49, 50, 51, 52]), open source hardware ([53, 25, 54]) RISC-V ([5, 55, 56, 57, 58, 59, 60, 61, 62, 63]), Network-on-Chips ([64, 65, 66, 67, 68]), security ([69, 70, 71, 72]), benchmark suites ([73, 74, 75]), dark silicon ([76, 77, 76, 78, 79, 80, 41, 44]),

multicore ([5, 81, 82, 83, 84, 67, 85, 86, 87, 68, 88, 89, 90, 91]), compiler tools ([92, 93, 94, 95, 96, 97, 98, 99, 100]) and FPGAs ([101, 102, 75]).

References

[1] RISC-V PLIC Specification, https://github.com/riscv/riscv-plic-spec/blob/master/riscv-plic.adoc.

[2] Dennard Scaling, https://en.wikipedia.org/wiki/Dennard_scaling.

[3] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, page 1–12, New York, NY, USA, 2017. Association for Computing Machinery.

[4] Cerebras Press Release, https://www.businesswire.com/news/home/20210420005955/en/Cerebras-

Systems-Smashes-the-2.5-Trillion-Transistor-Mark-with-New-Second-Generation-Wafer-Scale-Engine.

[5] D. Petrisko, F. Gilani, M. Wyse, D. C. Jung, S. Davidson, P. Gao, C. Zhao, Z. Azad, S. Canakci, B. Veluri, T. Guarino, A. Joshi, M. Oskin, and M. B. Taylor. BlackParrot: An Agile Open-Source RISC-V Multicore for Accelerator SoCs. *IEEE Micro*, pages 93–102, Jul/Aug. 2020.

[6] OpenTitan GitHub Repository, https://github.com/lowRISC/opentitan.

[7] Alex Forencich. The 1G Ethernet MAC module from verilog-ethernet GitHub Repository, https://github.com/alexforencich/verilog-ethernet/blob/master/rtl/eth_mac_1g_rgmii_fifo.v.

[8] lwIP - A Lightweight TCP/IP stack, https://savannah.nongnu.org/projects/lwip/.

[9] wolfSSL - An Embedded TLS Library, https://www.wolfssl.com/.

[10] The BlackParrot SDK, https://github.com/black-parrot-sdk/black-parrot-sdk.

[11] Liteeth Driver from Linux 5.15, https://github.com/torvalds/linux/blob/v5.15/drivers/net/ethernet/litex/litex_liteeth.c.

[12] RISC-V PLIC Driver from Linux 5.15, https://github.com/torvalds/linux/blob/v5.15/drivers/irqchip/irq-sifive-plic.c.

[13] The Yocto Project, https://www.yoctoproject.org/.

[14] OpenSBI GitHub Repository, https://github.com/riscv-software-src/opensbi.

[15] RISC-V SBI Specification, https://github.com/riscv-non-isa/riscv-sbi-doc.

[16] Dromajo, https://github.com/chipsalliance/dromajo.

[17] Shashank Vijaya Ranga. ParrotPiton and ZynqParrot: FPGA Enablements for the BlackParrot RISC-V Processor,
https://www.bsg.ai/papers/theses/Shashank_Thesis_Final.pdf.

[18] Zynq-7000 SoC,
https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html.

[19] Litex, https://github.com/enjoy-digital/litex.

[20] OSI Model, https://www.forcepoint.com/cyber-edu/osi-model.

[21] LiteEth GitHub Repository, https://github.com/enjoy-digital/liteeth.

[22] LeWiz GitHub Repository, https://github.com/lewiz-support/LMAC_CORE1.

[23] Zedboard, https://digilent.com/shop/zedboard-zynq-7000-arm-fpga-soc-development-board/.

[24] Avnet Network FMC Module, https://www.avnet.com/shop/us/products/avnet-engineering-services/aes-fmc-netw1-g-3074457345635205181/.

[25] Michael B. Taylor. BaseJump STL: SystemVerilog needs a Standard Template Library for Hardware Design. In *Design Automation Conference*, June 2018.

[26] RISC-V HLIC Driver from Linux 5.15,
https://github.com/torvalds/linux/blob/v5.15/drivers/irqchip/irq-riscv-intc.c.

[27] Ajay Brahmakshatriya, Emily Furst, Victor Ying, Claire Hsu, Max Ruttenberg, Yunming Zhang, Tommy Jung, Dustin Richmond, Michael Taylor, Julian Shun, Mark Oskin, Daniel Sanchez, and Saman Amarasinghe. Taming the zoo: A unified graph compiler framework for novel architectures. In *ISCA*, 2021.

[28] Emily Furst. *Code Generation and Optimization of Graph Programs on a Manycore Architecture*. PhD thesis, University of Washington, 2021.

[29] Xingyao Zhang, Haojun Xia, Donglin Zhuang, Hao Sun, Xin Fu, Michael Taylor, and Shuaiwen Leon Song. η-LSTM: Co-designing highly-efficient large lstm

training via exploiting memory-saving and architectural design opportunities. In *ISCA*, 2021.

[30] Chenhao Xie, Xie Li, Yang Hu, Huwan Peng, Michael Taylor, and Shuaiwen Leon Song. Q-VR: System-level design for future mobile collaborative virtual reality. In *ASPLOS*, 2021.

[31] D. Park, S. Pal, S. Feng, P. Gao, J. Tan, A. Rovinski, S. Xie, C. Zhao, A. Amarnath, T. Wesley, J. Beaumont, K. Chen, C. Chakrabarti, M. B. Taylor, T. Mudge, D. Blaauw, H. Kim, and R. G. Dreslinski. A 7.3 M Output Non-Zeros/J, 11.7 M Output Non-Zeros/GB Reconfigurable Sparse Matrix–Matrix Multiplication Accelerator. *IEEE Journal of Solid-State Circuits*, pages 933–944, April 2020.

[32] S. Pal, D. Park, S. Feng, P. Gao, J. Tan, A. Rovinski, S. Xie, C. Zhao, A. Amarnath, T. Wesley, J. Beaumont, K. Chen, C. Chakrabarti, M. Taylor, T. Mudge, D. Blaauw, H. Kim, and R. Dreslinski. A 7.3 M Output Non-Zeros/J Sparse Matrix-Matrix Multiplication Accelerator using Memory Reconfiguration in 40 nm. In *Symposium on VLSI Circuits*, pages C150–C151, 2019.

[33] Qiaoshi Zheng, Nathan Goulding-Hotta, Scott Ricketts, Steven Swanson, Michael Bedford Taylor, and Jack Sampson. Exploring energy scalability in coprocessor-dominated architectures for dark silicon. *Transactions on Embedded Computing Systems (TECS)*, Mar 2014.

[34] B. Beresini, S. Ricketts, and M.B. Taylor. Unifying manycore and fpga processing with the RUSH architecture. In *Adaptive Hardware and Systems (AHS), 2011 NASA/ESA Conference on*, pages 22 –28, 2011.

[35] Ganesh Venkatesh, John Sampson, Nathan Goulding, Sravanthi Kota Venkata, Michael Bedford Taylor, and Steven Swanson. QsCores: Configurable Co-processors to Trade Dark Silicon for Energy Efficiency in a Scalable Manner. In *International Symposium on Microarchitecture (MICRO)*, 2011.

[36] Jack Sampson, Manish Arora, Nathan Goulding-Hotta, Ganesh Venkatesh, Jonathan Babb, Vikram Bhatt, Michael Bedford Taylor, and Steven Swanson. An Evaluation of Selective Depipelining for FPGA-based Energy-Reducing Irregular Code Coprocessors. In *Conference on Field Programmable Logic and Applications (FPL)*, 2011.

[37] N. Goulding-Hotta, J. Sampson, G. Venkatesh, S. Garcia, J. Auricchio, P. Huang, M. Arora, S. Nath, V. Bhatt, J. Babb, S. Swanson, and M. Taylor. The GreenDroid Mobile Application Processor: An Architecture for Silicon's Dark Future. *Micro, IEEE*, pages 86–95, March 2011.

[38] Steven Swanson and Michael Taylor. GreenDroid: Exploring the next evolution for smartphone application processors. In *IEEE Communications Magazine*, March 2011.

[39] Manish Arora, Jack Sampson, Nathan Goulding-Hotta, Jonathan Babb, Ganesh Venkatesh, Michael Bedford Taylor, and Steven Swanson. Reducing the Energy Cost of Irregular Code Bases in Soft Processor Systems. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2011.

[40] Jack Sampson, Ganesh Venkatesh, Nathan Goulding-Hotta, Saturnino Garcia, Steven Swanson, and Michael Bedford Taylor. Efficient Complex Operators for Irregular Codes. In *High Performance Computing Architecture (HPCA)*, 2011.

[41] Nathan Goulding, Jack Sampson, Ganesh Venkatesh, Saturnino Garcia, Joe Auricchio, Jonathan Babb, Michael Taylor, and Steven Swanson. GreenDroid: A Mobile Application Processor for a Future of Dark Silicon. In *HOTCHIPS*, 2010.

[42] Nathan Goulding-Hotta. *Specialization as a Candle in the Dark Silicon Regime*. PhD thesis, University of California, San Diego, 2020.

[43] Moein Khazraee. *Specialization as a Candle in the Dark Silicon Regime*. PhD thesis, University of California, San Diego, 2020.

[44] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor.

Conservation cores: reducing the energy of mature computations. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[45] Michael Bedford Taylor, Luis Vega, Moein Khazraee, Ikuo Magaki, Scott Davidson, and Dustin Richmond. ASIC clouds: Specializing the datacenter for planet-scale applications. *CACM*, pages 103–109, 2020.

[46] Shaolin Xie, Scott Davidson, Ikuo Magaki, Moein Khazraee, Luis Vega, Lu Zhang, and Michael B. Taylor. Extreme datacenter specialization for planet-scale computing: Asic clouds. In *ACM Sigops Operating System Review*, 2018.

[47] Michael Bedford Taylor. Geocomputers and the Commercial Borg. In *SIGARCH Computer Architecture Today*, Dec 2017.

[48] Michael Taylor. The Evolution of Bitcoin Hardware. *Computer, IEEE*, Sept-Oct. 2017.

[49] Moein Khazraee, Luis Vega, Ikuo Magaki, and Michael Taylor. Specializing a Planet's Computation: ASIC Clouds. *IEEE Micro*, May 2017.

[50] Moein Khazraee, Lu Zhang, Luis Vega, and Michael Taylor. Moonwalk: NRE Optimization in ASIC Clouds or, accelerators will use old silicon. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[51] Ikuo Magaki, Moein Khazraee, Luis Vega, and Michael Taylor. ASIC Clouds: Specializing the Datacenter. In *International Symposium on Computer Architecture (ISCA)*, 2016.

[52] Michael B. Taylor. Bitcoin and the Age of Bespoke Silicon. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, 2013.

[53] Michael Bedford Taylor. Your agile open source HW stinks (because it is not a system). In *ICCAD*, 2020.

[54] Hadi Esmaeilzadeh and Michael Bedford Taylor. Open Source Hardware: Stone Soups and Not Stone Satues, Please. In *SIGARCH Computer Architecture Today*, Dec 2017.

[55] A. Rovinski, C. Zhao, K. Al-Hawaj, P. Gao, S. Xie, C. Torng, S. Davidson, A. Amarnath, L. Vega, B. Veluri, A. Rao, T. Ajayi, J. Puscar, S. Dai, R. Zhao, D. Richmond, Z. Zhang, I. Galton, C. Batten, M. B. Taylor, and R. G. Dreslinski. Evaluating Celerity: A 16-nm 695 Giga-RISC-V Instructions/s Manycore Processor With Synthesizable PLL. *IEEE Solid-State Circuits Letters*, 2(12):289–292, 2019.

[56] A. Rovinski, C. Zhao, K. Al-Hawaj, P. Gao, S. Xie, C. Torng, S. Davidson, A. Amarnath, L. Vega, B. Veluri, A. Rao, T. Ajayi, J. Puscar, S. Dai, R. Zhao, D. Richmond, Z. Zhang, I. Galton, C. Batten, M. B. Taylor, and R. G. Dreslinski. A 1.4 GHz 695 Giga Risc-V Inst/s 496-Core Manycore Processor With Mesh On-Chip Network and an All-Digital Synthesized PLL in 16nm CMOS. In *2019 Symposium on VLSI Circuits*, pages C30–C31, 2019.

[57] Scott Davidson, Shaolin Xie, Chris Torng, Khalid Al-Hawaj, Austin Rovinski, Tutu Ajayi, Luis Vega, Chun Zhao, Ritchie Zhao, Steve Dai, Aporva Amarnath, Bandhav Veluri, Paul Gao, Anuj Rao, Gai Liu, Rajesh K. Gupta, Zhiru Zhang, Ronald Dreslinski, Christopher Batten, and Michael Bedford Taylor. The Celerity Open-Source 511-core RISC-V Tiered Accelerator Fabric. *Micro, IEEE*, Mar/Apr. 2018.

[58] Ritchie Zhao, Chun Zhao, Shaolin Xie, Bandhav Veluri, Luis Vega, Christopher Torng, Ningxiao Sun, Austin Rovinski, Anuj Rao, Gai Liu, Paul Gao, Scott Davidson, Steve Dai, Aporva Amarnath, KhalidAl-Hawaj, Tutu Ajayi Christopher Batten, Ronald G. Dreslinski, Rajesh K.Gupta, Michael B.Taylor, and Zhiru Zhang. Celerity: An Open Source RISC-V Tiered Accelerator Fabric. In *7th RISC-V Workshop*, 2017.

[59] Tutu Ajayi, Khalid Al-Hawaj, Aporva Amarnath, Steve Dai, Scott Davidson, Paul Gao, Gai Liu, Atieh Lotfi, Julian Puscar, Anuj Rao, Austin Rovinski, Loai Salem,

Ningxiao Sun, Christopher Torng, Luis Vega, Bandhav Veluri, Xiaoyang Wang, Shaolin Xie, Chun Zhao, Ritchie Zhao, Christopher Batten, Ronald G. Dreslinski, Ian Galton, Rajesh K. Gupta, Patrick P. Mercier, Mani Srivastava, Michael Bedford Taylor, and Zhiru Zhang. Celerity: An Open Source RISC-V Tiered Accelerator Fabric. In *HOTCHIPS*, Aug 2017.

[60] Luis Vega and Michael Bedford Taylor. RV-IOV: Tethering RISC-V Processors via Scalable I/O Virtualization . In *CARRV*, 2017.

[61] Dai Cheol Jung. Caches for Complex Open Source System-on-Chip Designs. Master's thesis, University of Washington, 2019.

[62] Shashank Vijaya Ranga. ParrotPiton and ZynqParrot: FPGA Enablements for the BlackParrot RISC-V Processor. Master's thesis, University of Washington, 2021.

[63] Sripathi Muralitharan. TinyParrot: An Integration-Optimized Linux-Capable Host Multicore. Master's thesis, University of Washington, 2021.

[64] Dai Cheol Jung, Scott Davidson, Chun Zhao, Dustin Richmond, and Michael Bedford Taylor. Ruche Networks: Wire-Maximal, No-Fuss NoCs. In *NOCS*, 2020.

[65] Daniel Petrisko, Chun Zhao, Scott Davidson, Paul Gao, Dustin Richmond, and Michael Bedford Taylor. NoC Symbiosis. In *NOCS*, 2020.

[66] Yi Zhu, Michael Taylor, Scott B. Baden, and Chung-Kuan Cheng. Advancing supercomputer performance through interconnection topology synthesis. In *International Conference on Computer-Aided Design (ICCAD)*, pages 555–558, 2008.

[67] Michael B. Taylor, Walter Lee, Saman Amarasinghe, and Anant Agarwal. Scalar Operand Networks. In *IEEE Transactions on Parallel and Distributed Systems*, February 2005.

[68] Jason Kim, Michael B. Taylor, Jason Miller, and David Wentzlaff. Energy Characterization of a Tiled Architecture Processor with On-Chip Networks. In *International Symposium on Low Power Electronics and Design (ISLPED)*, August 2003.

[69] Sadullah Canakci, Leila Delshadtehrani, Furkan Eris, Michael Bedford Taylor, Manuel Egele, and Ajay Joshi. Directfuzz: Automated test generation for rtl designs using directed graybox fuzzing. In *DAC*, 2021.

[70] Byron Hawkins, Brian Demsky, and Michael Bedford Taylor. BlackBox: Lightweight Security Monitoring for COTS Binaries. In *Code Generation and Optimization*, 2016.

[71] Byron Hawkins, Brian Demsky, and Michael Bedford Taylor. A Runtime Approach to Security and Privacy. In *European Security and Privacy*, 2016.

[72] Alric Althoff, Joseph McMahan, Luis Vega, Scott Davidson, Timothy Sherwood, Michael Taylor, and Ryan Kastner. Hiding Intermittant Information Leakage with Architectural Support for Blinking. In *International Symposium on Computer Architecture (ISCA)*, 2018.

[73] Shelby Thomas, Chetan Gohkale, Enrico Tanuwidjaja, Tony Chong, David Lau, Saturnino Garcia, and Michael Bedford Taylor. CortexSuite: A Synthetic Brain Benchmark Suite. In *International Symposium on Workload Characterization (IISWC)*, Oct. 2014.

[74] Sravanthi Kota Venkata, IkkJin Ahn, Donghwan Jeon, Anshuman Gupta, and Michael Taylor. SD-VBS: The San Diego Vision Benchmark Suite. In *IEEE International Symposium on Workload Characterization (IISWC)*, 2009.

[75] Jonathan Babb, Matthew Frank, Victor Lee, Elliot Waingold, Rajeev Barua, Michael Taylor, Jang Kim, Srikrishna Devabhaktuni, and Anant Agarwal. The Raw Benchmark Suite: Computation Structures for General Purpose Computing. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 1997.

[76] Michael Taylor. A Landscape of the New Dark Silicon Design Regime. In *Design Automation and Test in Europe*, April 2014.

[77] Michael Taylor. A Landscape of the New Dark Silicon Design Regime. *Micro, IEEE*, Sept-Oct. 2013.

[78] Michael B. Taylor. Is Dark Silicon Useful? Harnessing the Four Horsemen of the Coming Dark Silicon Apocalypse. In *Design Automation Conference (DAC)*, 2012.

[79] Vikram Bhatt, Nathan Goulding-Hotta, Qiaoshi Zheng, Jack Sampson, Steve Swanson, and Michael B. Taylor. Sichrome: Mobile web browsing in Hardware to save Energy . In *Dark Silicon Workshop, ISCA*, 2012.

[80] Nathan Goulding-Hotta, Jack Sampson, Qiaoshi Zheng, Vikram Bhatt, Steven Swanson, and Michael Taylor. GreenDroid: An Architecture for the Dark Silicon Age. In *Asia and South Pacific Design Automation Conference (ASPDAC)*, 2012.

[81] Anshuman Gupta, Jack Sampson, and Michael Bedford Taylor. Qualitytime: A simple online technique for quantifying multicore execution efficiency. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014.

[82] Anshuman Gupta, Jack Sampson, and Michael Bedford Taylor. DR-SNUCA: An energy-scalable dynamically partitioned cache. In *International Conference on Computer Design (ICCD)*, 2013.

[83] Anshuman Gupta, Jack Sampson, and Michael Bedford Taylor. Time Cube: A Manycore Embedded Processor with Interference-Agnostic Progress Tracking. In *International Conference On Embedded Computer Systems: Architectures, Modeling And Simulation (SAMOS)*, 2013.

[84] Michael Taylor. *Tiled Microprocessors*. PhD thesis, Massachusetts Institute of Technology, 2007.

[85] Michael B. Taylor, Walter Lee, Jason Miller, David Wentzlaff, Ian Bratt, Ben Greenwald, Henry Hoffmann, Paul Johnson, Jason Kim, James Psota, Arvind Saraf, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay

Architecture for ILP and Streams. In *International Symposium on Computer Architecture (ISCA)*, June 2004.

[86] Michael B. Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffmann, Paul Johnson, Jae-Wook Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs. In *IEEE Micro*, March 2002.

[87] MB Taylor, J Kim, J Miller, F Ghodrat, B Greenwald, P Johnson, W Lee, A Ma, N Shnidman, V Strumpen, et al. The raw processor-a scalable 32-bit fabric for embedded and general purpose computing. In *Proceedings of Hot Chips XIII*, 2001.

[88] Michael B. Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffmann, Paul Johnson, Walter Lee, Arvind Saraf, Nathan Shnidman, Volker Strumpen, Saman Amarasinghe, and Anant Agarwal. A 16-issue Multiple-Program-Counter Microprocessor with Point-to-Point Scalar Operand Network. In *IEEE International Solid-State Circuits Conference (ISSCC)*, February 2003.

[89] Michael B. Taylor, Walter Lee, Saman Amarasinghe, and Anant Agarwal. Scalar Operand Networks. In *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, February 2005.

[90] Michael B. Taylor, Walter Lee, Saman Amarasinghe, and Anant Agarwal. Scalar Operand Networks: On-Chip Interconnect for ILP in Partitioned Architectures. In *International Symposium on High Performance Computer Architecture (HPCA)*, February 2003.

[91] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, and Anant Agarwal. Baring it all to Software: Raw Machines. In *IEEE Computer*, September 1997.

[92] Donghwan Jeon, Saturnino Garcia, and Michael Bedford Taylor. Skadu: Efficient Vector Shadow Memories for Poly-scopic Program Analysis. In *Conference on Code Generation and Optimization (CGO)*, 2013.

[93] S. Garcia, Donghwan Jeon, C. Louie, and M.B. Taylor. The Kremlin Oracle for Sequential Code Parallelization. *Micro, IEEE*, 32(4):42–53, July-Aug. 2012.

[94] Donghwan Jeon, Saturnino Garcia, Chris Louie, and Michael Bedford Taylor. Kismet: Parallel Speedup Estimates for Serial Programs. In *Conference on Object-Oriented Programming, Systems, Language and Applications (OOPSLA)*, 2011.

[95] Saturnino Garcia, Donghwan Jeon, Chris Louie, and Michael Bedford Taylor. Kremlin: Rethinking and Rebooting gprof for the Multicore Age. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 2011.

[96] Donghwan Jeon, Saturnino Garcia, Chris Louie, and Michael Bedford Taylor. Parkour: Parallel Speedup Estimates from Serial Code. In *USENIX Workshop on Hot Topics in Parallelism (HOTPAR)*, 2011.

[97] Donghwan Jeon, Saturnino Garcia, Chris Louie, Sravanthi Kota Venkata, and Michael Bedford Taylor. Kremlin: Like gprof, but for Parallelization. In *Principles and Practice of Parallel Programming (PPoPP)*, 2011.

[98] Saturnino Garcia, Donghwan Jeon, Chris Louie, Sravanthi Kota Venkata, and Michael Bedford Taylor. Bridging the Parallelization Gap: Automating Parallelism Discovery and Planning. In *USENIX Workshop on Hot Topics in Parallelism (HOTPAR)*, 2010.

[99] Karen Zee, Viktor Kuncak, Michael Taylor, and Martin C. Rinard. Runtime checking for program verification. In *RV*, 2007.

[100] A. Agarwal, S. Amarasinghe, R. Barua, M. Frank, W. Lee, V. Sarkar, D. Srikrishna, and M. Taylor. The RAW compiler project. In *Proceedings of the Second SUIF Compiler Workshop*, pages 21–23, 1997.

[101] Yi Zhu, Yuanfang Hu, Michael Taylor, and Chung-Kuan Cheng. Energy and switch area optimizations for FPGA global routing architectures. In *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, January 2009.

[102] Hu, Zhu, Taylor, and Cheng. FPGA Global Routing Architecture Optimization Using a Multicommodity Flow Approach . In *ICCD*, 2007.