THE UNIVERSITY OF QUEENSLAND

AUSTRALIA

Embedded Systems Super Thesis Project: FPGA RISC-V Softcore Processor for Network Security

Lachlan Comino

45321199

Universty of Queensland, 2024

School of Electrical Engineering and Computer Science, EECS

Abstract

This thesis explores the implementation and evaluation of a network security processor using RISC-V softcores on FPGA hardware, with particular focus on hardware-accelerated AES encryption. Using the VexRiscvSMP processor and the Digilent Arty A7 FPGA platform, we developed and compared single-core, dual-core, and quad-core configurations to determine optimal performance characteristics for encrypting network traffic. Custom AES instructions, created by the maintainers of VexriscvSMP, were implemented and integrated into LibreSSL that achieved a 4x speedup in raw AES operations compared to software implementations.

Performance analysis revealed significant bottlenecks in ethernet throughput and memory access patterns, particularly in multi-core configurations. The optimal design—an improved dual-core implementation running at 150MHz with expanded ethernet buffers—achieved a maximum throughput of 406 KB/s, representing an 84% improvement over the baseline single-core configuration. However, comparative analysis with a Raspberry Pi 4B showed that our FPGA implementation achieved only 1/40th of the throughput of conventional ARM-based systems, highlighting current limitations of FPGA-based softcore processors for high-performance network security applications.

This work demonstrates that while FPGA-based security processors are feasible, their practical application may be better suited to low-to-medium throughput scenarios where power efficiency and hardware-level security are prioritized over raw performance. The findings suggest that future improvements should focus on enhanced memory architectures, more cryptographic accelerators, and optimised network interfaces to better leverage the potential of FPGA-based security solutions.

Declaration by author

Lachlan Comino
l.comino@uq.edu.au
45321199

November 5, 2024

04/11/2024 Prof Michael Brunig
Head of School
School of Electrical Engineering and Computer Science
The University of Queensland
St Lucia, QLD 4072

Dear Professor Brunig,

This thesis is in accordance with the requirements of a Bachelor in Engineering (Honours), in the School of Electrical Engineering and Computer Science. I am submitting the following thesis to you, entitled:

“Embedded Systems Super Thesis Project: FPGA RISC-V Softcore Processor for Network Security”

This thesis was performed under the supervision of Matthew D’Souza. Additionally, I declare that all of the work presented in this thesis is my own, unless cited otherwise, and that it has not been previously submitted for a degree at UQ or any other institution.

Sincerely,
Lachlan Comino

Acknowledgments

Firstly, I would like to extend thanks to my supervisor, Matthew D'Souza, for his guidance and setting up the Raspberry Pi Cluster. All equipment used to undertake this project was from him.

Thank you also to all the volunteers who make the open-source RISC-V stack possible. Their extraordinary work across LiteX, Buildroot Linux, VexRiscvSMP and F4PGA made this project not only feasible but also served as an excellent demonstration of how complex SoC development can still have a minimal barrier to entry.

Additional thanks to those who offered help on the Zephyr Discord server. Although, we did not find a fix for the issue, the experience helped me deeply appreciate the inner-workings of Zephyr.

Finally, I wish to thank my friends and family for their unconditional support and encouragement during this project.

Contents

List of Figures

List of Tables

List of Abbreviations and Symbols

Abbreviations	
TCP	Transmission Control Potocol
IP	Internet Protocol
HTTP	Hypertext Transfer Protocol
FTP	File Transfer Protocol
TFTP	Trivial File Transfer Protocol
SSH	Secure Shell
DNS	Domain Name System
UDP	User Datagram Protocol
MAC	Media Access Control
Wi-Fi	Wireless Fidelity
PPP	Point-to-Point Protocol
SSL	Secure Sockets Layer
RISC-V	Reduced Instruction Set Architecture Five
ISA	Instruction Set Architecture
MTU	Maxium Transmission Unit
MII	Medium Independent Interface
SFP	Small Form-factor Pluggable
RCE	Remote Code Execution
AES	Advanced Encryption Standard
CBC	Cipher Block Chaining
TLS	Transport Layer Security

Chapter 1

Introduction

1.1 Motivation

The explosive growth of the Internet of Things (IoT) has led to an unprecedented increase in internet-connected devices, with estimates suggesting over 75 billion IoT devices by 2025 [13]. However, this growth has brought to attention the importance of network security in embedded systems, especially where sensitive data is handled. Cyber-attacks targeting IoT devices have become more sophisticated and frequent, with the number of IoT attacks increasing by 300% in 2019 alone [14].

The 2016 Dyn incident is perhaps the most sophisticated example of an IoT attack, where attackers compromised thousands of insecure IoT devices and successfully targeted Dyn, a DNS provider, with a total data stream of 1.2Tb/s, effectively taking down major portions of the internet. While this happened eight years ago and Cloudflare promises that DDOS attacks of this scale can now be mitigated at a high-level[1], the reality is that thousands of IoT devices, such as security cameras, smart home appliances, sensor devices *etc.*, were freely accessible due to a lack of security.

Furthermore, traditional software-based security approaches often struggle to keep up with the real-time requirements and resource constraints of IoT devices [15]. Frustaci et al. [15] emphasize that the limited memory, processing power, and energy resources of IoT devices make it challenging to implement strong security measures using software alone without compromising performance and battery life.

This is where a RISC-V-based SoC could show promise. Although, RISC-V processors may not yet match the raw performance of established ARM or x86 architectures, the ISA has in recent years gained significant commercial validation. Most notably, NVIDIA's recent announcement detailed plans to ship over a billion RISC-V cores by 2025 [16], accomodating for "virtually all MCU cores" in their GPUs [16]. This move highlights a key advantage of RISC-V where its open-source nature eliminates licensing fees and royalties that typically burden proprietary architectures.

[1] https://www.cloudflare.com/en-au/ddos/

1.2 Project Aims

This thesis will create a case for leveraging Field Programmable Gate Arrays (FPGAs) and RISC-V to create an optimised network security solution. By utilising the flexibility of FPGAs and RISC-V's extensible architecture, it becomes possible to implement security features at the hardware level while maintaining the performance requirements of modern IoT applications.

This project will aim to achieve the primary objectives:

1. Fasten existing network security software through hardware-accelerated encryption.
2. Cheap implementation of hardware-acceleration, regarding FPGA utilisation.
3. Low-latency in packet processing and network operations.
4. Power-efficiency, compared to software-based solutions.
5. And scalablibility to accommodate additional common embedded systems peripherals.

Additionally, the use of open-source tools for FPGA development will be used and evaluated.

1.3 Project Scope

This project focuses on the implementation and evaluation of a RISC-V softcore processor with integrated network security features, specifically targeting FPGA deployment. The scope encompasses:

1. Development of a multi-core RISC-V processor implementation using the VexRiscv architecture.
2. Integration of hardware-accelerated AES encryption through custom instructions.
3. Implementation of essential networking capabilities including ethernet interfacing.
4. Comprehensive performance evaluation against both single-core and multi-core configurations.
5. Comparison with conventional software-based security implementations.

Due to the vast, ever-changing nature of the field of cyber-security, The project will not address:

- Physical security measures or tamper resistance.
- Side-channel attack prevention.
- Security protocols beyond the ones that use AES encryption.
- Operating system-level security features.

Through this scope, we are aiming to demonstrate the effectiveness of hardware-level security implementation, while maintaining system performance within the constraints of FPGA resources.

Chapter 2

Literature Review

2.1 Computer Networking

Communication between devices on a network is enabled through a standardised set of protocols and interfaces, allowing devices to communicate regardless of any underlying hardware or architecture. The TCP/IP protocol suite is most common set of protocols which organises these protocols into distinct layers to handle different aspects of network communication. This layered approach allows for modular development and maintenance of network functionality [1] [2] [17].

There are five primary layers. Figure 2.1, shows how these layers are organised.

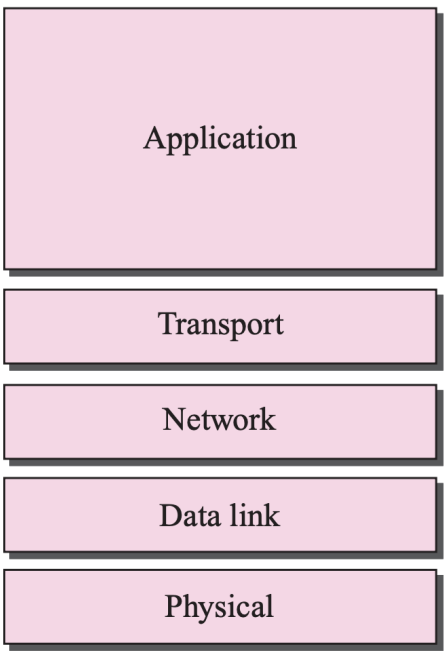


Figure 2.1: TCP/IP Model [1]

4. Application Layer:

Provides network services directly to end-users. Protocols include HTTP, FTP, SSH, and DNS.

3. Transport Layer:

Manages end-to-end communication. Implements TCP and UDP transfer protocols. Handles flow control and errors such as missing packets at a high-level.

2. Network Layer:

Routes packets between networks using IP addressing. Handles packet fragmentation and reassembly. Makes routing decisions and uses best-effort delivery without guarantees.

1. Link Layer:

Manages direct communication between devices on the same network. Handles physical addressing via MAC. Also provides error detection. Protocols include Ethernet, Wi-Fi, and PPP.

0. Physical Layer:

Responsible for transmitting raw bits over physical media (copper wire, fiber optic or radio). Electrical signal methods are handled here, such as voltage levels for HIGH and LOW bits, and timing.

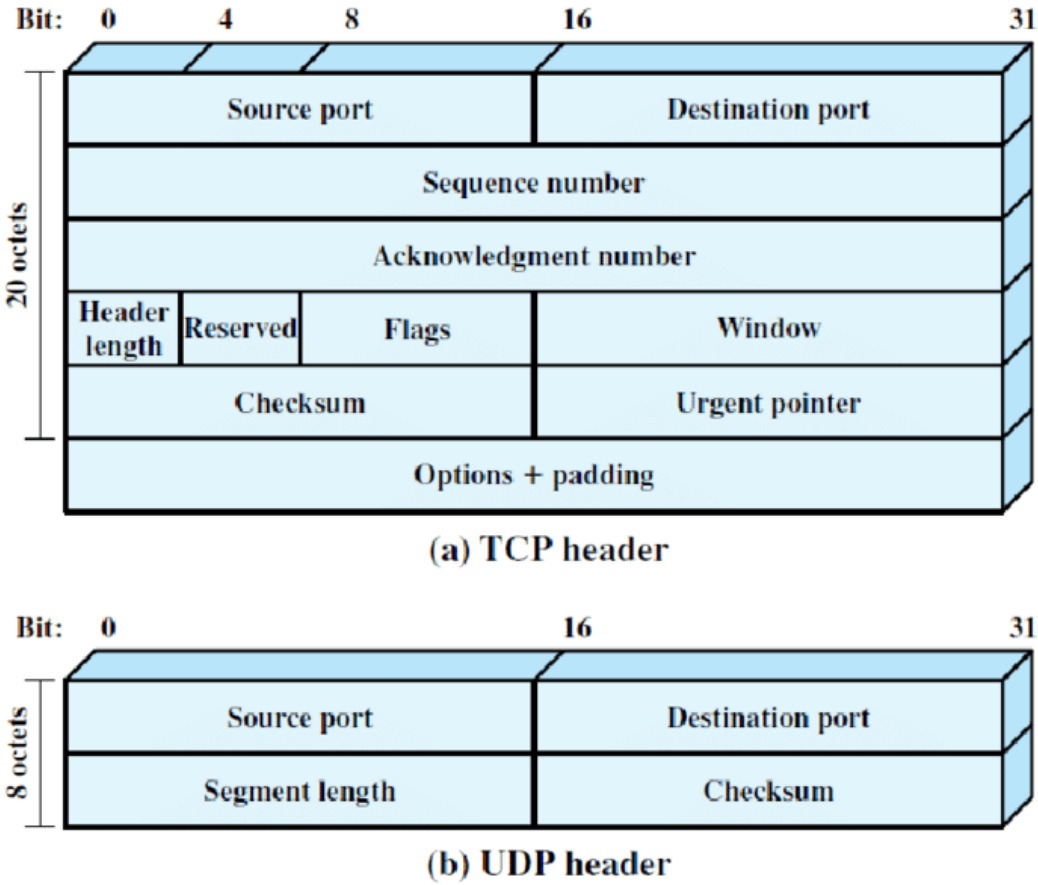
In the coming sections, we will detail each of the relevant layers in the project and the important protocols in each. It should be noted that the technologies and protocols spanning the TCP/IP stack are vast and extensively documented; the following sections serve only as a rudimentary summary of the concepts most pertinent to this project's implementation.

2.1.1 Transport Layer

The transport layer provides end-to-end communication services between applications running on different hosts. At its core, this layer segments application data into smaller units called packets. A packet consists of a header containing protocol-specific control information, and a payload containing the actual data being transmitted [2].

In terms of how these packets are handled, we have two over-arching protocols which distinctly handle the two most prominent types of data transfer. For lossless transmission and reliability at the cost of a performance overhead, we have the Transmission Communication Protocol, TCP. For connectionless transfers where only data throughput is prioritised, we have User Datagram Protocol. The idea behind each protocol is best demonstrated in the contrast between how packets are formatted, a visual representation of which can be seen in figure 2.2:

Figure 2.2: TCP and UDP headers [2]



TCP vs UDP

Before a TCP data transmission begins, connection through a three-way handshake between sender and receiver is established. This connection maintains state information throughout the transmission process, enabling tracking of every packet sent and received. For reliability, each packet is assigned a sequence number, allowing the receiver to reconstruct the data stream in the correct order and detect any missing packets. When packets are lost or corrupted, TCP's acknowledgment system automatically triggers retransmission [17].

UDP takes a different approach, operating as a connectionless protocol with minimal overhead. Unlike TCP, UDP begins transmitting data immediately without establishing a connection or maintaining state information [17]. It simply sends packets (called datagrams in UDP) to the target destination without guaranteeing their arrival or ordering. This means UDP cannot guarantee reliable delivery, but it achieves lower latency and higher throughput compared to TCP [17].

Sockets and Ports

Seen in Figure 2.2, both headers for UDP and TCP contain source/destination port fields. The Transport layer implements the concept of ports and sockets in order to multiplex network connections on a single device [1]. A socket, representing the combination of an IP address and 16-bit port number, creates a unique communication endpoint, which is an abstraction that allows for multiple applications to communicate simultaneously without conflict.

2.1.2 Network Layer

The Network layer handles the routing and addressing of packets between networks using the Internet Protocol (IP). For devices on a local network switch, as in this project, the Network layer's role is relatively straightforward - it primarily handles packet routing through IP and basic packet fragmentation when necessary [17]. Since all project devices exist within the same network segment connected via an ethernet switch, there is no complex routing involved; packets are simply addressed between known IP addresses on the local subnet, *i.e.*, no device will be separated by more than two links. The layer does, however, still maintain its responsibility for packet fragmentation and reassembly, ensuring data can be transmitted within the Maximum Transmission Unit (MTU) size of the local network [1].

2.1.3 Data Link Layer, Ethernet MAC

The Data Link layer, implemented through the Ethernet Media Access Control (MAC), manages direct communication between devices on the same network segment. The MAC sublayer handles addressing via unique 48-bit MAC addresses, frame formation, and error detection through Cyclic Redundancy Checks (CRC). Additionally, there is another field which specifies the "EtherType". EtherType is a 16-bit identifier that indicates which protocol is encapsulated in the frame's payload (e.g., 0x0800 for IPv4, 0x0806 for ARP, 0x86DD for IPv6). This field enables the receiver to correctly demultiplex

incoming frames to the appropriate higher-layer protocol handler in the networking stack. Figure 2.3 illustrates the structure of an Ethernet II frame. This frame type is most common, contains all the information needed to transfer data between ethernet MACs.

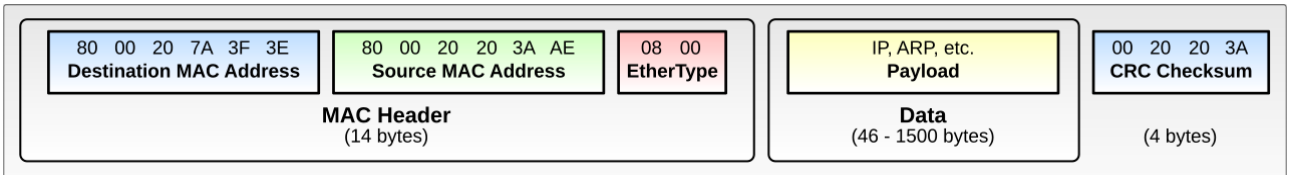


Figure 2.3: The Common Ethernet Frame Format, Type II Frame Buffer [3]

The MAC also implements carrier sense multiple access with collision detection (CSMA/CD) for shared medium arbitration, though this function is less utilised in modern switched networks where full-duplex operation is standard [17].

Ethernet MAC implementations on FPGAs typically use vendor-provided IP cores that implement the MAC layer in FPGA fabric. Xilinx offers several "soft" MAC implementations through Vivado, each with different trade-offs. The Tri-Mode Ethernet MAC (TEMAC) IP core supports 10/100/1000 Mbps operation with full features [18]. The AXI 1G/2.5G Ethernet Subsystem IP provides the same MAC functionality but with an AXI4-Stream interface for easier system integration [19]. Both these implementations can get exhaustive with the consumption of FPGA fabric resources such as Look-Up Tables (LUTs) and Block RAM (BRAM) for packet buffering. To mitigate this, if high-speed transfers are not required, the Ethernet Lite MAC IP offers a minimal resource implementation at 10/100 Mbps [20].

2.1.4 Physical Layer, Ethernet PHY

The Physical layer is responsible for the actual transmission of raw bits across the physical medium, typically through twisted-pair copper cables in modern Ethernet networks. At this layer, digital data from above is converted into electrical signals suitable for transmission.

The connection between the MAC and PHY is standardised through the Media Independent Interface (MII). This interface exists in several variants that support different speeds:

- MII: The standard interface supporting 10/100 Mbps using a 4-bit data path
- RMII (Reduced MII): A simplified interface using a 2-bit data path to reduce pin count
- GMII (Gigabit MII): An enhanced interface supporting up to 1000 Mbps with an 8-bit data path

For higher throughput applications beyond 1 Gbps, interfaces such as XGMII (10 Gbps) and CGMII (100 Gbps) exist. These are often implemented through Small Form-factor Pluggable (SFP) transceivers - modular interfaces that support both optical and copper connections at high speeds.

Complete lists of these ethernet interfaces are provided by the corresponding *Wikipedia* articles: MII [21] and SFP [22].

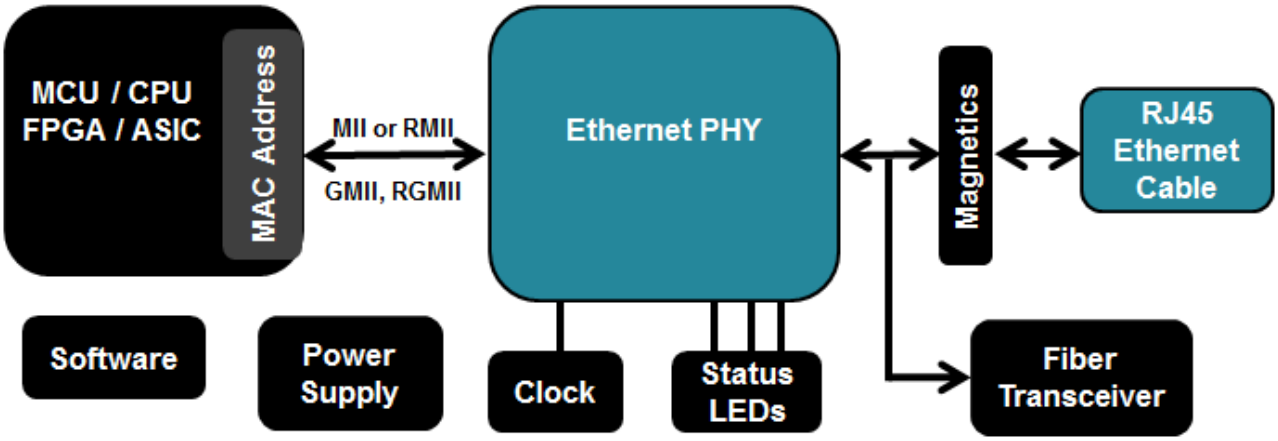
To attach a basic PHY interface, the Management Data Input/Output (MDIO) provides the serial communication channel between the MAC and PHY. Beyond just configuration and status monitoring, this two-wire interface (MDC clock and MDIO data) handles the essential data exchange between the MAC and PHY, including the speed negotiation, link status, and error conditions.

In FPGA implementations, achieving maximum throughput requires consideration of several factors. Clock domain crossing between the MAC and PHY interfaces must be properly managed, along with appropriate sizing of transmit/receive buffers. Most important, however, is the processing capability of the system itself - insufficient processing speed is guaranteed to bottleneck a higher-end ethernet interface.

Data Link/Physical Layer Summary

We mentioned several different interfaces in sections: 2.1.3 and 2.1.4, that carry an ethernet signal from the port to the CPU. Figure 2.4 is a high-level overview of a typical implementation, from an RJ-45 ethernet cable (right) to the device (left):

Figure 2.4: Ethernet PHY System Block Diagram [4]



2.2 Network Security

As more applications rely on IoT edge-computing, it is critical that engineers ensure that computer networks still remain secure. Unfortunately, the more we protocols we implement, especially at the hardware-level, the more complicated our applications become; straining edge computers which are most efficient when they are only single-purpose (as seen in [23] and [24]).

Additionally, it's not only network transfers where vulnerabilities can exist. Physical security measures must also be considered, as many embedded systems are deployed in accessible locations. Although unlikely, it is possible for dedicated hackers to create intrusions and RCEs based on memory-tampering [25]. Such attacks have been around since the conception of the C programming language,

Shao *et al.*shows how these attacks can be mitigated at a higher-level with intrusion detection [26], but ultimately, the onus is on developers to ensure that all low-level accesses to memory are consistently safe; *i.e.*, there are no possible edge cases that can trigger buffer overflows [25].

We can address these issues, and still have robust security posture for networked embedded systems, however, to be implemented properly, regular reviews and updates are also required as the field of cybersecurity is constantly adapting to new vulnerabilities. Due to the vast scope, this thesis will only focus on secure transfers rather than memory protection.

2.2.1 Transport Layer Security (TLS)

Transport Layer Security (TLS) operates in an additional layer, the Presentation layer (OSI model), between the Transport and Application layers of the TCP/IP stack. As defined in RFC 8446 [27], TLS ensures three crucial security properties: confidentiality through encryption, integrity via Message Authentication Codes (MACs, not to be confused with Media Access Control (MAC)), and authentication using digital certificates.

The complete explanation of TLS 1.3 is out of scope for this thesis, but for a basic summary, TLS begins with a handshake phase where communicating parties negotiate protocol versions and cryptographic algorithms, authenticate identities, and establish shared secret keys. After this handshake, the TLS Record Protocol handles the bulk encryption of application data, breaking it into blocks, encrypting it with the negotiated algorithms, and adding integrity checks before transmission [27]. Essentially, TLS ensures that no sensitive data is transferred without proper authorisation and end-to-end encryption. Figure 2.5 shows this exchange in addition to the TCP stream creation.

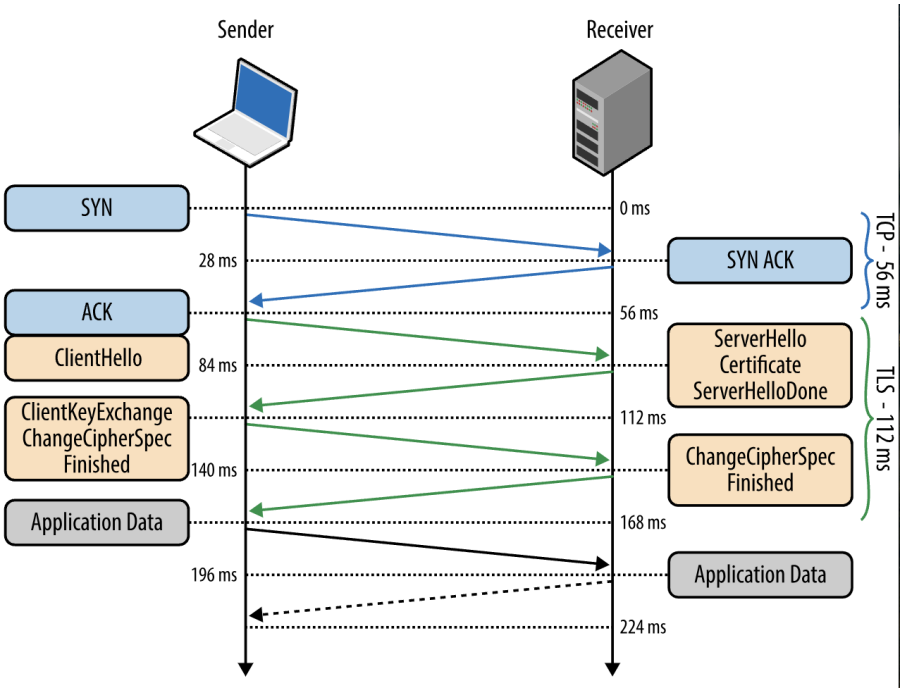


Figure 2.5: TLS and TCP Exchange Diagram [5]

TLS Stacks in Embedded Systems

The computational overhead of TLS in IoT contexts has been extensively studied. Isobe *et al.* [23] demonstrated that even with dedicated FPGA acceleration, achieving 10Gbps throughput requires significant architectural considerations including parallel processing circuits, shared transmission/reception pathways, and optimised switch designs to reduce wire overhead. Their implementation, while achieving high performance (10Gbps), required significant FPGA resources - approximately 48,000 slices for the complete TLS stack and 23W of constant power. Keep in mind as well, this work from 2010 predates modern TLS 1.3, which introduces its own set of performance considerations and security limitations [27].

For resource-constrained IoT devices, Restuccia et al. [24] revealed that the choice of cryptographic operations drastically impacts energy consumption. Their measurements showed that while TLS with Pre-Shared Keys (PSK) and AES-128-CCM consumed only 2.3 millicoulombs of charge during a complete handshake, upgrading to Elliptic Curve Diffie-Hellman (ECDHE) with ECDSA, for example, increased this to 63.4 millicoulombs - a factor of nearly 28x. They also demonstrated that moving from AES-128-CCM to AES-256-GCM increased memory requirements significantly, with heap usage growing from 5.7KB to 20.6KB for TLS 1.2 [24].

2.2.2 AES Encryption

This thesis will focus on efficiently implementing one such security consideration, albeit a critical one, which is encryption. Specifically, we will be implementing the Advanced Encryption Standard (AES), which has become the de-facto standard for symmetric key encryption in computer security.

At its core, AES is a block cipher that operates on fixed-size blocks of data (128 bits) using cipher keys of 128, 192, or 256 bits. The encryption process consists of multiple rounds of several processing steps, all of which aim to obfuscate the input data, yet also maintain reversibility for decryption. Here is a high-level overview of the specific steps to perform an AES encryption, paraphrased from the *Wikipedia* article, [6].

1. *KeyExpansion* – round keys are derived from the cipher key. AES uses a separate 128-bit round key block for each round plus one more.
2. *AddRoundKey* - each byte of the state is combined with a byte of the round key using bitwise XOR. This function is responsible for incorporating the key into encryption process.

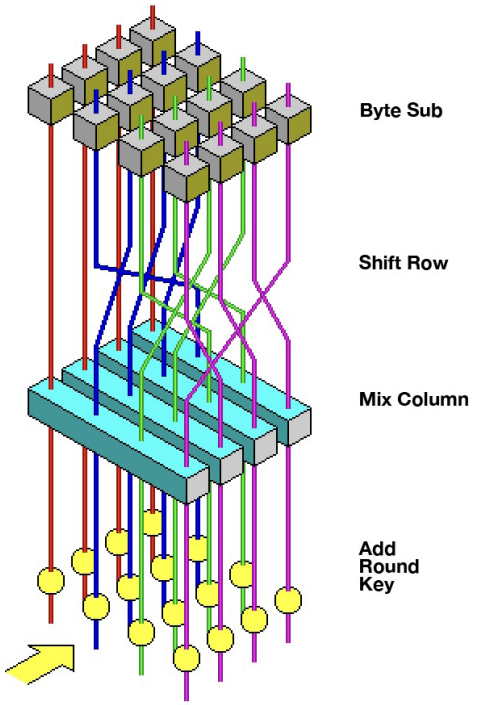


Figure 2.6: AES Round Function [6]

3. Then the middle rounds are executed:
   1. *SubBytes*: each byte is substituted with another according to a lookup table (the SBox).
   2. *ShiftRows*: then, the last three rows of the substituted bytes are shifted cyclically.
   3. *MixColumns*: The four bytes in each column are then combined.
   4. *AddRoundKey*: finally, the round key is XOR-ed with each byte of the state (same as step 2).

These middle rounds are executed n times, depending on the key size:

- $n = 10$ rounds for 128-bit keys
- $n = 12$ rounds for 192-bit keys
- $n = 14$ rounds for 256-bit keys

The final output is our encrypted input text. AES has been heavily and publicly scrutinised for decades since its adoption by the U.S. government [28] (aside from side-channel attacks, not technically a fault of the algorithm). It's also performant, as it can be implemented in-place with constant-time operations. And it's flexible, as multiple key-sizes are available and can be chosen based on an application's memory/time complexity constraints. However, due to the sequential nature of the CBC, CCM and GCM, AES variants, parallelisation, *i.e.*processing blocks in parallel, is not a viable option for accelerating proper encryption; since each block is used in the following round.

It's worth mentioning too that the decryption process essentially reverses the encryption process; only it uses inverse versions of the encryption steps, applied in a reverse order.

2.2.3 Performing an AES encryption via OpenSSL

A single AES 256-bit CBC encryption of a text file can be performed via the open-source, OpenSSL library which implements SSL and TLS protocols for linux systems [29]:

‘‘‘ `openssl enc -aes-256-cbc -salt -in plaintext.txt -out encrypted.bin -k MySecretKey -pbkdf2` ’’’

Where:

- `-aes-256-cbc`: Specifies the encryption algorithm (AES-256 in CBC mode).
- `-salt`: Adds a salt to the password to prevent specific attacks, (dictionary and rainbow tables attacks) [29].
- `-in plaintext.txt`: The input file to be encrypted.
- `-out encrypted.bin`: The output file where the encrypted data will be stored.
- `-k MySecretKey`: The key used for encryption.
- `-pbkdf2`: Uses PBKDF2 (Password-Based Key Derivation Function 2) for key generation, which is more secure than the default (A warning for deprecated key generation usage will appear if this is omitted (OpenSSL 3.3.1)).

The ‘`plaintext.txt`’ file is only securely encrypted if the key is not made public.

2.3 RISC-V ISA Technical Overview

2.3.1 Core Design Philosophy

RISC-V distinguishes itself from other Instruction Set Architectures (ISAs) through its fundamental design principles of modularity, simplicity, and extensibility. Unlike complex instruction set computing (CISC) architectures such as x86, which have accumulated thousands of instructions over decades of development, RISC-V begins with a minimal base integer instruction set (RV32I or RV64I) and uses a fixed-width 32-bit instruction format [11]. This base ISA remains fixed for compatibility while allowing architectural evolution through optional extensions, a key difference from both x86's variable-length instructions (1-15 bytes) and traditional RISC architectures like MIPS or SPARC that had relatively fixed ISAs [30]. The architecture mandates that all processors must implement this base instruction set (CSR access and control is located here as well), but allows selective implementation of both standard extensions (denoted by letters like 'M' for integer multiplication, 'F' for floating-point) and custom extensions, enabling developers to create application-specific processors [31].

2.3.2 Standard Extensions

RISC-V's modular design allows implementors to selectively include standard extensions based on their application requirements. Each extension is denoted by a single letter, with additional modifiers for variant specifications. Table 2.1 outlines the instruction extensions that are currently supported by the chosen softcore processor, VexRiscv (a softcore 32-bit processor), which will be described later in section 2.5. Rows that are bold in Table 2.1, are the chosen instruction sets that are implemented in the bitstream, since a key advantage of RISC-V's extensibility in FPGA implementations is the ability to omit unused features, saving on FPGA resources.

Extension	*Name*
M	**Integer Multiplication/Division**
A	**Atomic Instructions**
F	Single-Precision Float
C	Compressed Instructions

Table 2.1: Common Risc-V Extensions in Embedded Systems and Descriptions

The M (Integer Multiplication and Division) extension adds multiply, divide, and remainder operations to the base integer instruction set. It's a critical extension for general-purpose computing, providing both full-width (MUL, DIV, REM) and reduced-width operations (MULH, DIVU, REMU) that operate on integers [11].

The Atomic (A) extension is particularly important for multicore implementations, providing atomic memory operations that are adapted for inter-core synchronisation. These instructions, such as atomic-add (AMOADD) and atomic-swap (AMOSWAP), perform read-modify-write operations in a single, uninterruptible sequence, ensuring thread-safe memory access in concurrent systems [11].

The Compressed (C) extension is notable, as it's designed to improve code density and thus memory usage. It provides 16-bit versions of common 32-bit instructions, enabling up to a 30% reduction in code size [31]. The extension achieves this by encoding frequently used instruction patterns with shorter formats, such as common register-to-register operations or small immediate values. This can be valuable in applications where memory is limited [32]. But of course, using compressed instructions does come with a constant performance overhead from decoding.

Finally, floating-point extensions (F for 32-bit, D for 64-bit and Q for 128-bit precision), we have omitted, since floating-point operations aren't required in the application. This saves on DSP blocks and logic elements but relateively, they are inexpensive to implement compared to the significant boost in floating point calculations they provide.

2.3.3 ISA-Level Security Features

Outlined in volume two of the RISC-V Manual [10], are several architectural features that increase security at the ISA level. To reiterate, hardware-level security is not the objective of this thesis, but it's still worth mentioning since these features are simple to generate using *LiteX* and are supported by *Zephyr* and *Buildroot Linux* (we will cover these in their respective sections). For instance, The

privilege mode system is one such feature, it divides instructions into three levels: Machine (M), Supervisor (S), and User (U) modes.

Level	Encoding	Name	Abbreviation
0	00	User/Application	U
1	01	Supervisor	S
2	10	*Reserved*	
3	11	Machine	M

Table 2.2: RISC-V privilege levels [10].

Machine mode has the highest privileges and is typically used for critical system functions, Supervisor mode enables operating system functionality like virtual memory management, while User mode runs application code with restricted access to system resources (the ISA extensions from Section 2.3.2 are implemented at the User level) [10]. This hierarchical approach ensures strict isolation between different software components.

Additionally, Physical Memory Protection (PMP) allows Machine mode to set memory access permissions and restrictions for lower privilege modes, enabling additional control over memory regions. PMP can prevent unauthorised access to sensitive memory areas and is particularly valuable in embedded systems where memory protection is crucial [10]. Once again, these features support a strong case for the usage of RISC-V in security, but ultimately it is out of the project's scope. Research does exist however, in the effectiveness of these security approaches; for example, Lu *et al.* [33].

2.3.4 Implementing Custom Instructions

We have discussed minimising/expanding the instruction set, but it is also possible to implement your own custom instructions. RISC-V reserves custom opcode spaces that exist in the G extension (general-purpose) [11], see ''$custom-0/1$' in Table 2.3. Each of these spaces can accomodate multiple 32-bit instructions.

inst[4:2] / inst[6:5]	000	001	010	011	100	101	110	111 ($> 32b$)
00	LOAD	LOAD-FP	*custom-0*	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	$48b$
01	STORE	STORE-FP	*custom-1*	AMO	OP	LUI	OP-32	$64b$
10	MADD	MSUB	NMSUB	NMADD	OP-FP	*reserved*	*custom-2/rv128*	$48b$
11	BRANCH	JALR	*reserved*	JAL	SYSTEM	*reserved*	*custom-3/rv128*	$\geq 80b$

Table 2.3: RISC-V base opcode map, inst[1:0]=11 [11].

Custom instructions can be added by defining the instruction format (typically following RISC-V's standard R, I, S, or U-type formats), allocating opcode fields, and implementing the corresponding execution logic. The instruction is then integrated into the processor pipeline, usually in the decode and execute stages. These instructions can then be used most-primitively via inline assembly in C.

2.4 FPGAs

Field Programmable Gate Arrays (FPGAs) are integrated circuits designed to be configured by customers after manufacturing. Unlike ASICs, FPGAs provide flexibility through their reprogrammable nature, making them ideal for creating custom hardware.

The architecture of an FPGA consists of a grid-like arrangement of configurable logic blocks distributed across a silicon die, connected by a programmable interconnect network. This grid structure, often called the ”fabric”, contains several fundamental building blocks [34]:

- Look-Up Tables (LUTs): Combinational logic functions.
- Flip-Flops: Store state information and implement sequential logic.
- Block RAM (BRAM): Provides on-chip memory resources.
- DSP Slices: Digital signal processing operations.
- Programmable Interconnects: Connect logic blocks via configurable routing.

And the price of an FPGA chip is largely dependent on the amount of these building blocks.

Digital circuits are implemented on FPGAs through a process called synthesis, where hardware descriptions (written in HDL or higher-level languages) are transformed into configurations of these basic building blocks. The synthesis tools (such as Xilinx’s *Vivado*), handle the complex tasks of mapping logic to physical resources and routing connections between them [34].

2.4.1 Softcore CPUs

Softcore processors are CPU cores implemented using the programmable logic resources of an FPGA. Unlike hardcore processors that are physically implemented in silicon, softcore CPUs offer customization flexibility at the cost of lower performance [34]. Their configurability makes them particularly valuable in specialised FPGA applications where sequential processing is required.

Currently, there’s no shortage of RISC-V cores to choose from, according to this list [35]. In terms of performance comparison, there is a blog article created in 2020 (presumably outdated by now), by *Just Another Electronics Blog* [12] which profiles significant variations among popular cores in terms of performance, resource utilization, and features. VexRiscv notably achieves a balance between performance (0.75 DMIPS/MHz) and resource usage (769/665 LUT/FF), but their full findings are presented in Table 2.4.

It’s worth mentioning, due to the nature of softcore processors, there two inherent limitations compared to regular hardcore CPUs, some can be considered debilitating, depending on the application:

1. Timing Constraints: Logic elements in FPGAs are physically separated, which introduces more propagation delay compared to regular CPUs [36]. This is presumably why RISC-V softcore processors rarely exceed 250MHz.

2. Power Efficiency: The general-purpose FPGA fabric consumes more power than specialised silicon, making softcore CPUs less energy-efficient than their hardcore equivalents [36].

    - Softcore CPUs also do not support a variable clock rate for matching idle or straining loads. They run at a constant clock rate and thus are always at full power.

	Vexriscv	LEON3	PicoRV32	NEO430	ZPUFlex	Microwatt
Ease of use	+	+	o	++	+	o
Debugger	Yes	Yes	No	Bootloader	No	No
Documentation	o	++	-	++	o	-
Adding peripherals	+	+	+	+	o	+
DMIPS/Mhz	0.75 (1.57)	0.84 (1.4)	0.2 (0.51)	0.15	0.06	0.5
Size CPU core LUT/FF	769/665	4542/1552	1000/701	420/126	369/189	7337/3465
Clockspeed	127Mhz	85Mhz	170Mhz	100Mhz	215Mhz	100Mhz
DMIPS @ max speed	95	71	34	15	12.9	50
Language	SpinalHDL	VHDL	Verilog	VHDL	VHDL	VHDL
License	MIT	GPL/Commercial	ISC	BSD	?	CC-BY 4.0

Table 2.4: Popular Risc-V Cores Performance and Implementation Comparision, 2020 [12]

Despite these constraints, the open-source community still continues to push the boundaries of RISC-V softcore processor design.

2.5 VexRiscv

VexRiscv[1] represents a unique approach to RISC-V implementation through its highly modular plugin architecture. Unlike traditional CPU designs that use fixed pipeline stages with hardcoded functionality, VexRiscv employs a plugin system where virtually every CPU feature - from the instruction decoder to the register file - is implemented as a plugin [37]. This allows for extreme configurability which is valuable for resource management in FPGAs.

The core pipeline consists of 2 to 5+ stages [37], with optional plugin features like: MMU support for Linux compatibility, debug capabilities for GDB integration, custom instruction extensions, hardware FPU support, *etc.*. To get an overview of how feature count affects performance and FPGA resource utilisation, the repository itself, has a comparison[2].

Additionally, VexRiscv is implemented in SpinalHDL, an abstraction language built on top of Scala (similar to how *TypeScript* adds type safety to *JavaScript*). SpinalHDL leverages Scala's object-oriented and functional programming features to abstract away Verilog's low-level implementation

[1] Github Repository: https://github.com/SpinalHDL/VexRiscv

[2] Area usage and maximal frequency: https://github.com/SpinalHDL/VexRiscv/blob/master/README.md#area-usage-and-maximal-frequency

details [38]. This higher-level approach is what makes VexRiscv's plugin architecture possible by allowing developers to write complex, modular hardware descriptions that are then automatically converted to synthesisable Verilog.

2.5.1 VexRiscvSMP

VexRiscvSMP extends the base VexRiscv design to enable multiple CPU cores to work together in parallel, in a process called "Symmetric Multi-Processing". For a brief explanation of how the multiple cores use a shared memory and peripherals, an extra interfacing layer is implemented [37] [1], known as a MESI (Modified, Exclusive, Shared, Invalid) cache coherency protocol with three key interfaces:

1. Read Interface: Allows cores to obtain new memory copies and make them unique when they need to write to them. For example, when a core needs to write to memory that other cores might have cached.
2. Write Interface: Used by cores to write data back to main memory and notify the system when they no longer need their cached copies.
3. Probe Interface: Allows the system to manage cache coherency by instructing cores to change their cache states or provide their data when another core needs it.

Now with this interfacing layer, cores can communicate with memory and peripherals, the same way as the standard Vexriscv, through a Wishbone bus, which handles the physical connection. This entire system ensures that when multiple cores access shared resources (like UART or memory), their operations remain coordinated and data remains consistent. This explains the memory-management aspect of SMP, however, a second primary component known as an interrupt controller is also utilised for sharing interrupts from peripherals and inter-core communication (known as IPI scheduling (Inter-Processor-Interrupts)). This component is external to the VexRiscvSMP architecture and will be elaborated on in Section 2.6.2.

2.6 Litex

LiteX is an open-source SoC builder framework that overcomes the creation of complex FPGA designs by providing a Python-based (Migen), infrastructure for digital system integration. Unlike traditional hardware development flows that require manual RTL integration and bus interconnection, LiteX automates many of the tedious aspects of SoC design. It provides a collection of pre-built cores (UART, Timer, RAM, Ethernet), standardised bus interfaces (Wishbone, AXI, Avalon), and support for various softcore processors, including VexRiscv.

Figure 2.7 shows a high-level overview of a Linux-capable SoC generated by LiteX. Note that the AXI bus is also connected to additional whitespace labelled, "*User Design*", showing that if the FPGA resources allow, additional modules and FPGA designs can be added.

[1] Documentation: https://github.com/SpinalHDL/VexRiscv/blob/master/doc/smp/smp.md

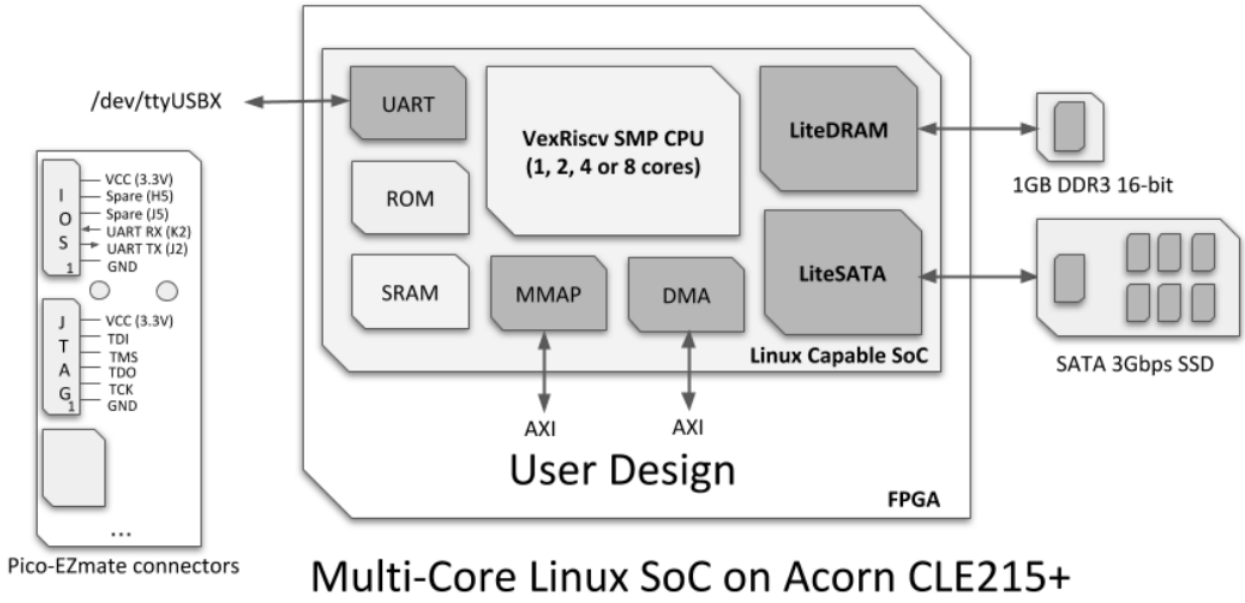


Figure 2.7: SoC for Running Linux on an Acorn CLE215+ [7]

The framework supports major FPGA vendors (Xilinx, Intel, Lattice) and can integrate components written in different HDLs (VHDL, Verilog, SpinalHDL), making it highly versatile for different development needs. Overall, the automation and flexibility provided by LiteX significantly reduces development time [7].

2.6.1 Litex Cores

Prefixed by "Lite", LiteX provides a comprehensive set of IP cores that handle various system functionalities. LiteDRAM serves as the memory controller, supporting DDR2/DDR3/DDR4 SDRAM with configurable parameters and automatic PHY calibration.

For networking, LiteEth manages ethernet communication through support for 10/100/1000 PHYs, customizable buffer sizes, and hardware CRC checking. Storage access is handled by cores such as LiteSPI and LiteSDCard, which provide memory-mapped interfaces to their respective mediums with support for various transfer modes and automatic chip select handling. Additional cores include LitePCIe for PCIe endpoint/root port implementation, and LiteScope which provides an integrated logic analyzer for debugging (similar to simulation in *Vivado*). Each core is written in Migen/Python, making them portable across different FPGA platforms.

LiteX also has existing support for common vendor-specific features like Xilinx's Internal Configuration Access Port (ICAP) for runtime bitstream reconfiguration and XADC for voltage/temperature monitoring.

In our implementation, we primarily use LiteDRAM for the board's 256MB DDR3 module, LiteEth configured with 8KB TX/RX buffers for the 100Mbps ethernet and LiteSPI for the 16MB Quad-SPI

flash. These cores are responsible for exposing the functionality provided by the Digilent Arty A7 evaluation board (see Section 3.1.2).

2.6.2 SiFive PLIC/CLINT

In a multi-core system, managing interrupts from various peripherals and coordinating between cores presents a conundrum. For instance, how do cores share access to a single ethernet peripheral, or how does one core signal another for task scheduling? LiteX addresses this through two standard RISC-V interrupt controllers based on the SiFive design: the Platform-Level Interrupt Controller (PLIC) and Core-Local Interruptor (CLINT) [8].

The PLIC manages external interrupt sources from peripherals by routing to appropriate cores and prioritisation through 7 levels of priority (7 being the highest). The CLINT handles core-local interrupts including software interrupts (IPIs) and timer interrupts, which are essential for multi-core synchronisation and task scheduling [8].

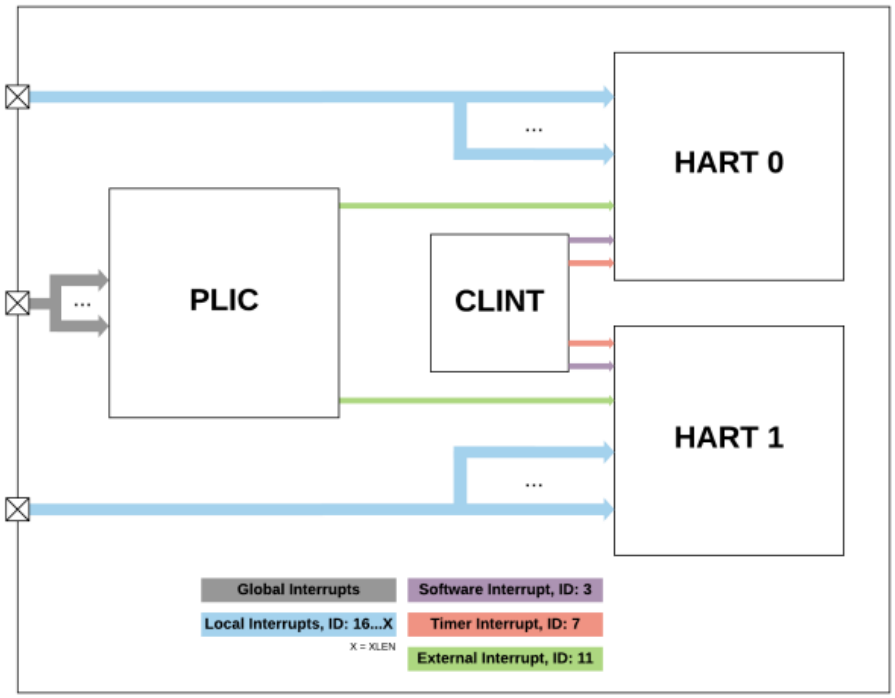


Figure 2.8: PLIC + CLINT PLIC Block Diagram for Machine Mode [8]

As shown in Figure 2.8, each HART (hardware thread, representing an execution context in RISC-V), has dedicated connections to both controllers, with the PLIC mapping global interrupts to external device interrupts (ID: 11) and the CLINT managing software (ID: 3) and timer (ID: 7) interrupts. Both controllers are scalable to accommodate varying numbers of cores and interrupt sources, with memory-mapped registers for configuration and control.

The PLIC can occupy up to 4MB of address space for interrupt configuration registers, theoretically supporting up to 1024 unique interrupt sources, while the CLINT can use up to 64KB for software

interrupts and timer comparators, capable of mapping to thousands of HARTs. Essentially, the SiFive PLIC/CLINT enables coordinated interrupt handling across all cores, with routing and prioritisation [8].

2.6.3 LiteX BIOS

A key feature of LiteX that emphasises its ease of use, is the LiteX BIOS (Basic Input/Output System). This is a built-in firmware that is flashed to your SoC alongside the loading of the FPGA bitstream. It provides essential system initialization and debugging capabilities:

1. System Identification: Reports the CPU type, bus configuration, and memory map of the system.
2. Memory Initialisation: Configures and tests the system memory (SDRAM), verifying both write and read capabilities through memtests.
3. Boot Options: Provides flexibility in booting from various sources (serial, SPI flash, *etc.*).

If no boot medium is provided, the BIOS will default to the LiteX console. Here, several immediate functions are provided that help test different parts of the system, such as an I2C scanner for detecting I2C devices (if I2C is a part of the SoC). The console can be accessed through a simple serial connection to the FPGA board [7]. See Appendix A.6 for an example of full BIOS output.

2.6.4 SoC Generation with LiteX

To complement our theoretical discussion of LiteX's capabilities, we will now examine its practical development workflow. The development process for LiteX centers around board target files - Python scripts that describe hardware using Migen DSL. These files are coupled with Python's argparse library to make hardware descriptions configurable from the command line. For example, the Arty A7 target file contains the board's basic platform definition (pins, clocks, peripherals), but through command-line arguments, users can configure the specs of the softcore CPU or any of the peripherals. Here is a command-line argument that builds an example SoC for the Arty A7, assuming all dependencies are installed:

```
./digilent_arty.py --cpu-type neorv32 --with-spi-sdcard --with-pmod-gpio
--with-can --sdcard-adapter=numato --sys-clk-freq=50e6 --with-led-chaser
--integrated-main-ram-size=8192 --build
```

This command configures:

- Minimal RISC-V CPU (NEORV32)
- SPI-based SD card interface
- PMOD GPIO access
- Automotive CAN bus support
- Numato SD card adapter
- 50 MHz system clock
- LED animation module
- 8KB integrated block RAM

The build produces two main outputs: a 'gateware' directory containing FPGA synthesis products, and a 'software' directory containing the BIOS and support files. To load this to the FPGA, you will need a bootloader. For simplicity, it is recommended to use *openFPGALoader* [39]. Loading the bitstream is as simple as running:

```
openFPGALoader -b arty_a7_35t build/arty_a7/gateware/arty_a7.bit
```

Then to access the BIOS, just screen the USB port. For this, *picocom*, was used. It has more features than the default Ubuntu "screen" command:

```
picocom -b 115200 /dev/ttyUSB1 --imap lfcrlf
```

Upon each screen into the port, the FPGA will reboot, allowing you to see the full boot process, similar to Appendix A.6.

To start a new LiteX project, developers typically copy and modify an existing target file from the litex-boards repository, adapting it for their specific needs. Also, there are specialised repositories, built on top of LiteX, intended for generating SoCs that can run Zephyr [40] or Buildroot Linux [41].

2.6.5 Buildroot Linux

Buildroot provides a simplified framework for generating embedded Linux systems, making necessary compromises to run on resource-constrained devices [42]. Unlike standard Linux distributions that prioritise flexibility and feature-richness, Buildroot creates minimal, purpose-built systems by statically linking applications and using BusyBox, a space-efficient re-implementation of common Linux utilities, as well as core embedded system features like network stacks, filesystem utilities, and system initialisation [43].

For example, while a standard Linux distribution might include separate vi, grep, and ps binaries totaling several megabytes, BusyBox combines simplified versions of these tools into a single binary often under 1MB. Similarly, it replaces the complex systemd init system with a basic init process that sequentially executes startup scripts.

The build process involves selecting system components through a menuconfig interface, similar to the Linux kernel. However, Buildroot extends this concept across the entire system build, handling not just the kernel configuration but also the toolchain, root filesystem, and package selection. Configuration through menuconfig is hierarchical and interdependent. Selecting a particular package might automatically enable required dependencies or kernel features. For instance, enabling OpenSSL support would pull in necessary cryptographic kernel modules and certificate management tools. This dependency management ensures a coherent system build while maintaining minimal size.

When targeting a RISC-V system, the build process generates five essential components:

- A compressed root filesystem (.cpio).
- The Linux kernel image (Image).

- A compiled device tree blob (.dtb), which describes the hardware configuration.
- The OpenSBI firmware image (opensbi.bin) providing the supervisor binary interface.
- A boot.json file that the LiteX BIOS uses to locate the above files.

These components work together in a specific sequence during system startup: The LiteX BIOS loads OpenSBI through an available boot medium (UART, Ethernet, SPIFlash or SD card), which then initialises the hardware and sets up the supervisor mode environment for secure booting (RISC-V supervisor privilege, see Section 2.2). OpenSBI then loads the Linux kernel and device tree (Image and .dtb files), with the kernel mounting the root filesystem (.cpio) to complete the bootloading process. Once the filesystem and Linux kernel are initialised in memory, Linux will then begin to boot. The full output of this process can be seen in Appendix A.6.

OpenSBI (Supervisor Binary Interface) serves as the firmware layer between RISC-V hardware and the operating system [44], exactly like how x86 systems rely on BIOS/UEFI firmware for hardware initialisation. OpenSBI takes a more minimal approach, however, by implementing only the essential firmware services: CPU initialisation, trap handling, and device tree parsing. This minimalist design aligns with RISC-V's philosophy of simplicity, providing a clean separation between hardware and software layers while remaining flexible enough to support different platform implementations.

Chapter 3

Stack Overview

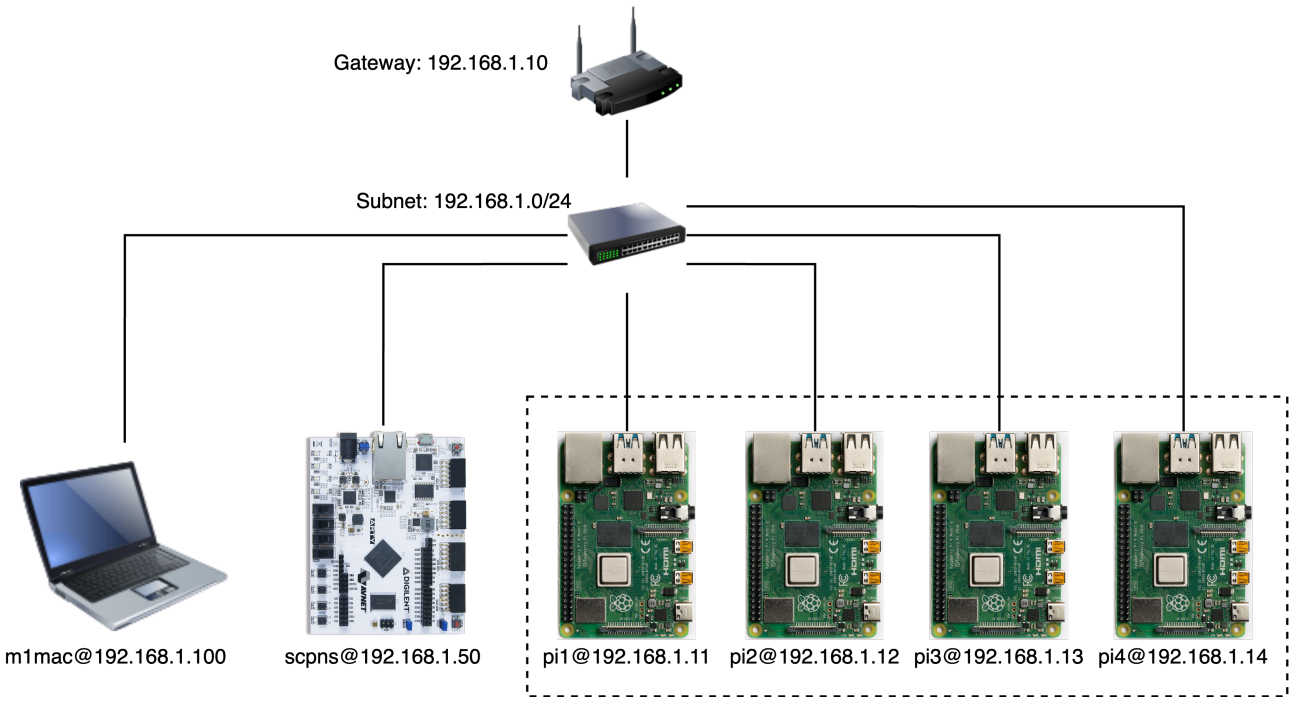
This section will now pertain to the complete stack overview of the proposed solution, starting with all of the hardware.

3.1 Hardware Setup

3.1.1 Network Overview

The network infrastructure, illustrated in Figure 3.1, will be a controlled environment designed to evaluate the performance of the FPGA. The network is made of three segments: one gateway router, a local area network, and six end devices.

Figure 3.1: Overview of the Devices in the Networking Application



For the end devices, there are three categories in terms of roles. A development workstation (m1mac@192.168.1.100), will do test orchestration, metrics collecting and TFTP (Trivial File Transfer Protocol). The FPGA-based security processor will be called the SCPNS for shorthand (Softcore Processor for Network Security), on scpns@192.168.1.50, and will serve as an encryption server, basically handling encryption/decryption jobs. Lastly, a cluster of four Raspberry Pi devices (pi1-pi4, addresses: .11-.14) will act as clients generating test workloads for the FPGA. The five other end devices will all be sending metrics to the workstation on 192.168.1.100.

3.1.2 FPGA Board

In this project, we will be using the Digilent Arty A7 35T, which is an evaluation board for the Xilinx Artix-7 FPGA (XC7A35). Figure 3.2 is an overview of the functions on the board. Fortunately, this board is one of the default recommended boards for LiteX, all of the external IO, Flash and RAM modules are supported.

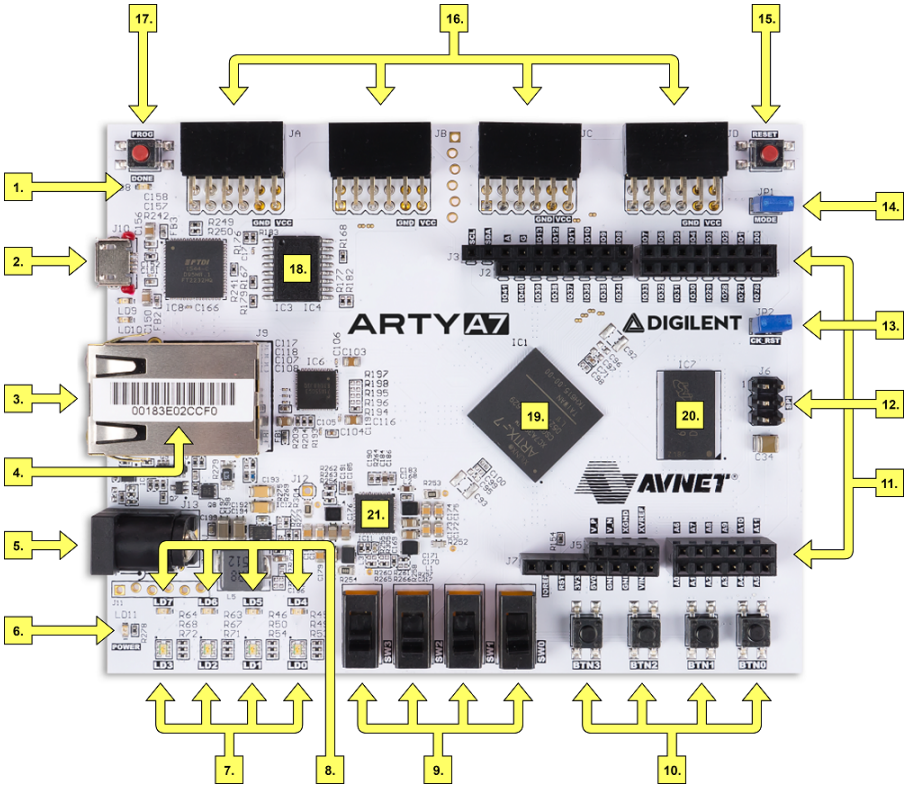


Figure 3.2: Labelled Arty A7 Board [9]

Notable features include the 100Mbps ethernet (3), a 256MB DDR3 external RAM module (20), 16Mb Flash (18) and PMOD expansion ports (16). It is not labelled on the diagram, but there is also an onboard FTDI FT2232HQ USB-UART bridge (located between the micro USB port (2) and the flash (18)), which allows for serial communication and JTAG programming/debugging through a single USB connection.

As for the FPGA chip itself, Table 3.1 shows all of the specifications for the Artix-7:

Table 3.1: Artix-7 35T Specifications

Resource/Feature	**Specification**
Part Number	XC7A35T-1CSG324C
Logic Slices: - LUTs (4 per Slice) - FFs (8 per Slice) - F7 Muxes (2 per Slice) - F8 Muxes (1 per Slice)	5,200 20,800 41,600 10,400 5,200
Block RAM	1,800 Kb (50 * 36 Kb blocks)
DSP Slices	90
Maximum Frequency	464 MHz
Clock Management Tiles	5
Processor Technology	28nm CMOS
Temp. Operating Range	0°C to 85°C

Additionally, on the Artix-7 there is an XADC (Xilinx Analog-to-Digital-Converter), which has a temperature sensing and external power monitoring readable via registers in the CSR. We will be using this to measure temperature throughout testing, and monitor power usage.

3.1.3 VexriscvSMP Configurations

Using LiteX, we have generated three different configurations for VexriscvSMP: a single-core, dual-core and quad-core. The idea is that via testing, we can determine if increasing cores results in increased performance for the SMP system. Typically, more cores is beneficial but only for raw CPU performance. If there is only one access point, say buffers in RAM for a certain resource, this can introduce contention, netting us diminishing returns for performance by the number of cores.

Each configuration will maintain identical specifications and full access to the board's peripherals:

- Custom AES instructions, detailed in Section 3.1.4.
- A RV32I ISA with M and A extensions (RV32IMA).
- DDR3PHY for accessing on-board DRAM module (LiteDRAM Core).
- Quad-SPI access to the Flash module.
- 4KB L1 cache per core, one 8KB shared L2 cache.
- Linux support in the form of cache coherency and memory management (explained in Section 2.5.1).
- Initialised surrounding peripherals, this includes: I2C, SPI, UART, XADC, Ethernet (Static IP), two MMCMs, switches, leds (including RGB) and buttons.

This consistent architecture ensures that performance differences can be attributed solely to the variation in core count. The memory layout for each of these configurations can be found in Figure 3.2.

Region Name	Address and Size	Physical Instantiation
OPENSBI	0x40f00000 0x80000	256MB DDR3 RAM Module
PLIC CLINT ROM SRAM	0xf0c00000 0x400000 0xf0010000 0x10000 0x00000000 0x10000 0x10000000 0x4000	1,800KB BRAM
MAIN_RAM	0x40000000 0x10000000	256MB DDR3 RAM Module
ETHMAC ETHMAC_RX ETHMAC_TX CSR	0x80000000 0x2000 0x80000000 0x1000 0x80001000 0x1000 0xf0000000 0x10000	1,800KB BRAM

Table 3.2: Memory Layout for VexRiscvSMP Linux and Physical Location.

3.1.4 Custom AES Instructions

The full implementation of these instructions, including the SpinalHDL code for the physical implemention in VexRiscvSMP, the C header drivers and LibreSSL patch (in Section 3.2.1), were all created by Charles Papon in 2021 [37]. All of these files regarding the implementation of the 'AesPlugin' can be found in the Vexriscv GitHub repository [37].

Building upon the AES algorithm described in Section 2.2.2, we will now implement hardware acceleration through custom RISC-V instructions. As discussed in Section 2.3.2, RISC-V allows custom instruction extensions through reserved opcode spaces, specifically the $custom-0$ (0001011) opcode space in this implementation. The instructions are encoded by default as following:

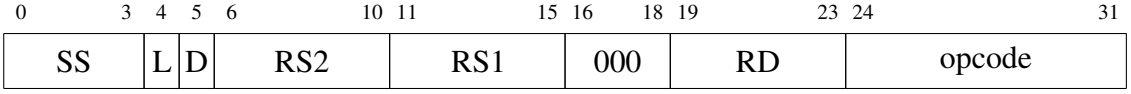


Figure 3.3: AES Instruction encoding

Where:

- SS: Selects which byte from RS2 to process (0-3).
- L: Distinguishes between standard round (0) and last round (1).
- D: Selects encryption (0) or decryption (1).
- RS2: Source register 2, general-purpose register containing key data.
- RS1: Source register 1, general-purpose register containing input data.
- RD: Destination register for storing result.

The data operated on by these registers is loaded from DRAM through separate load/store instructions before and after the AES instruction execution. See Appendix A.21 for how these instructions are defined in in-line RISC-V assembly in C.

Optimisation and Implementation

On our 32-bit RISC-V implementation (where a word is 32 bits - the natural data size the processor works with), we implement the standard AES optimisation that combines *SubBytes*, *ShiftRows*, and *MixColumns* into a single operation [6], through FPGA resources (most importantly some dedicated BRAM). This optimisation uses a 512-word ROM organised in two banks as:

1. **Bank 0** (0x000-0x0FF), Forward transformations:
    - Combines SBox lookup with *MixColumns* multiplication.
    - Each word contains {SBox*1, SBox*2, SBox*3, InvSBox}.
2. **Bank 1** (0x100-0x1FF), Inverse transformations:
    - Pre-computed inverse SBox with $GF(2^8)$ multiplications.
    - Each word contains {InvSBox*14, *9, *13, *11}.

So when we combine multiple SBox operations, multiplications, and inverse operations into single ROM (Read-Only Memory) lookups, we're essentially pre-computing these complex substitution and arithmetic steps to save processing time.

Now, to summarise, here is the full outline of steps that the RISC-V CPU undergoes when it fetches an AES instruction:

1. Selects a byte from RS2 based on the SS field
2. Uses this byte to address the appropriate ROM bank
3. Permutes the ROM output based on the SS field and instruction type
4. XORs the result with RS1
5. Writes the final value to RD

And the process for a single 128-bit block cipher is complete.

Hardware Implementation

Derived from synthesising with and without, the AES instructions were determined to consist of an additional 824 LUTs (as logic), 194 FFs and 256 F7 Muxes, contributing to an additional 1.5% utilisation.

3.1.5 Raspberry Pis

Lastly, acting as physical client devices, we have a set of four Raspberry Pis. Each Pi is a model 4B with 1GB RAM. This model features a quad-core ARM Cortex-A72 processor running at 1.8GHz, integrated Gigabit Ethernet, and uses a 32GB SD card for storage. They all will be running Ubuntu Server 22.04 LTS images (we do not need a desktop). This configuration provides a consistent test platform for generating network loads, but also, we will be able to see the SCPNS's encryption performance against a conventional ARM-based SBC [45].

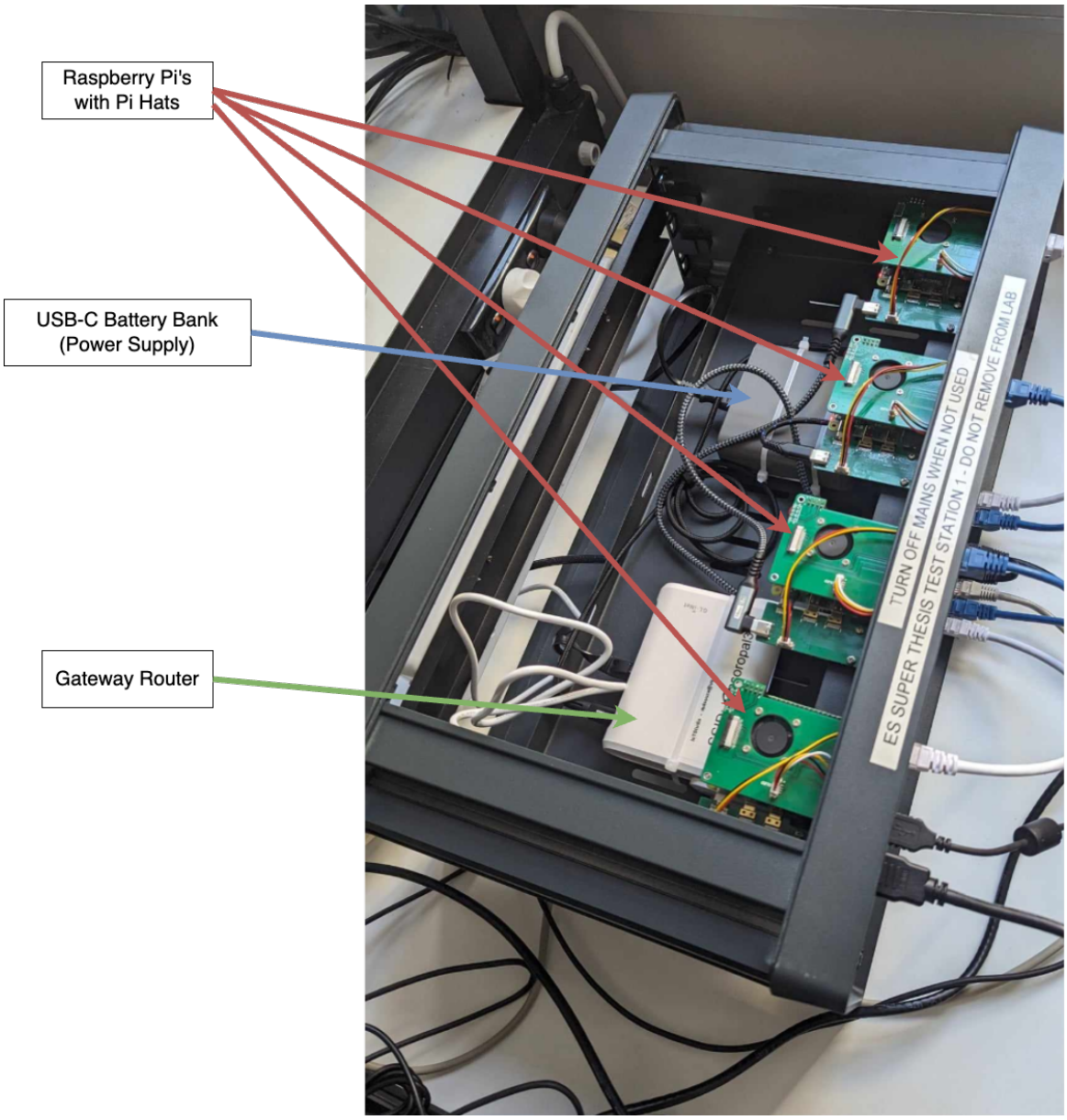


Figure 3.4: Annotated Photo of RPi Cluster. The network switch is the grey box underneath the router and power supply.

3.2 Software Setup

The complete software stack consists of the software running on the end devices which is a custom client-server application where multiple clients can request for encryption/decryption services from the SCPNS. The SCPNS will be able to handle multiple threads of these clients simultaneously, due to the SMP capability and the '*Threading*' Python standard library.

3.2.1 Buildroot Linux Configuration

The SCPNS runs a compiled Buildroot Linux image, with additional packages enabled via the menuconfig. These packages can be sorted by the purpose they achieve:

Benchmarking:

- Dhrystone - standardised CPU benchmark.
- Coremark - another CPU benchmark.
- Stress-ng - benchmarking for multi-core performance, IO and memory accession.
- Whetstone - floating point performance benchmark.
- iPerf3 - TCP throughput benchmarking.

Application-Specific:

- Python3
- OpenSSL & LibreSSL

Python Libraries:

- PSutil - for getting CPU and memory usage.
- SQLite - for storing metrics.

These additional packages increase the base prebuilt Linux filesystem from approximately 5MB to 33.8MB. The Buildroot Makefile also provides graphing targets to produce plots on different compile-time metrics. For instance, Figure 3.5 shows the size of each package in our distribution.

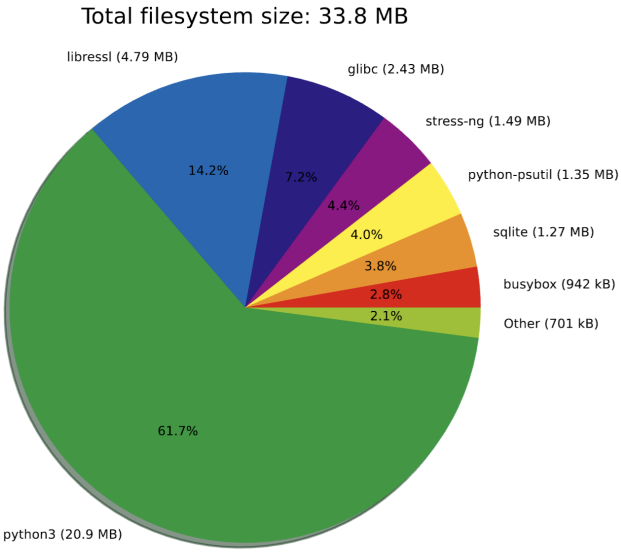


Figure 3.5: Size of Filesystem Packages.

LibreSSL Patch

To make software at the operating system layer use our custom AES instructions we can patch the AES function calls to point to our new instructions. In this case, we have patched LibreSSL's core AES implementation to use our custom instructions, enabling hardware acceleration for all the system's cryptographic operations, especially in TLS. This means any application using LibreSSL's TLS stack (like HTTPS or secure sockets) automatically benefits from our custom AES instructions without requiring modifications to application code.

Briefly, LibreSSL is a fork of OpenSSL created in 2014 by the OpenBSD project. It focuses on providing a more secure and simplified codebase, compared to OpenSSL, which can be ideal for embedded systems. The patch targets LibreSSL version 3.2.2, which aligns with the Buildroot Branch, 2023.2.x and provides the necessary hooks for hardware acceleration.

The patch modifies LibreSSL's AES implementation `aes_core.c`, and adds two new C header files: `aes_custom.h` and `riscv.h`, to leverage the custom AES instructions. The following is a breakdown of the patch:

1. **AES Implementation Replacement**, in `aes_core_vexriscv.c`:
    - Instantiates the originally software-based table lookups (Te0-Te4) in BRAM ROM.
    - Maintains the same API (*i.e.*`AES_encrypt()`, `AES_decrypt()`, key setup functions *etc.*) for compatibility.
    - Inherits functions from `aes_custom.h` and `riscv.h`.
2. **Hardware Interface**, `aes_custom.h` is a wrapper of the custom instruction defined in `riscv.h` (See Figure, 3.3). In `aes_custom.h`, there are two primary functions which execute the full encryption or decryption process, `vexriscv_aes_encrypt()` and `vexriscv_aes_decrypt()`, as well as surrounding formatting and memory alignment functions.
3. **Build System Integration:** Adds `VEXRISCV_AES` option to the crypto CMakeLists.txt (and by extension, the Buildroot defconfig), to allow us to enable or disable the patch at compile-time.

Now our hardware acceleration can be introduced at the library level without disrupting the existing software stacks.

LibreSSL Benchmarks

We can see if the instructions were implemented correctly by using OpenSSL's built-in speed test:

```
openssl speed -elapsed -evp aes-128-cbc aes-256-cbc
```

Testing raw encryption performance without memory I/O bottlenecks, both AES-128-CBC and AES-256-CBC show significant improvements with hardware acceleration. The custom instructions provide approximately 4x speedup for both variants at larger block sizes, with AES-128-CBC achieving 4.24 MB/s and AES-256-CBC reaching 3.39 MB/s (at 8192-byte blocks). See Table 3.3.

Block Size (bytes)	AES-128-CBC (No HW) (KB/s)	AES-128-CBC (HW) (KB/s)	AES-256-CBC (No HW) (KB/s)	AES-256-CBC (HW) (KB/s)
16	569.82	751.23	492.58	1065.34
64	879.49	1569.70	700.22	2242.60
256	1035.23	3435.13	796.41	3100.24
1024	1096.36	4146.81	831.49	3165.50
8192	1091.36	4246.01	830.09	3391.24

Table 3.3: OpenSSL Benchmarks Table

3.2.2 Application: Server and Client Encryption/Decryption

The application aims to evaluate how processor core count affects encryption throughput by implementing a multi-threaded encryption server that can handle simultaneous client requests. The server application runs on the SCPNS, utilizing the full Python standard library available through Buildroot Linux (*i.e.*, not MicroPython). This grants us robust thread management, network programming, and subprocess handling capabilities, typical of regular Python.

We will be making a basic OpenSSL subprocess call to utilise our hardware-accelerated AES instructions through the LibreSSL patch. For testing purposes, we use pre-computed key and initialisation vector values:

```
openssl enc -aes-128-cbc -K "00000000000000000" -iv "00000000000000000"
-nosalt [-nopad]
```

The -nopad flag is used for all chunks except the final one to ensure proper CBC block alignment in the TCP stream. Note that while using key/IV pairs consisting of all zeroes would be insecure in production, it allows us to focus purely on gathering encryption performance metrics.

The server implements client handling through Python's ThreadPoolExecutor, maintaining up to 10 concurrent client connections with one processing thread per connected client. This limit was chosen, not because the processor could not handle more, but because if the client count exceeded 10, there would be significant slowdown due to resource contention around the ethernet buffers and the main RAM module. So, instead, the additional clients are queued (using a thread-safe 'Queue'), to allow for faster reconnectivity once another client exits.

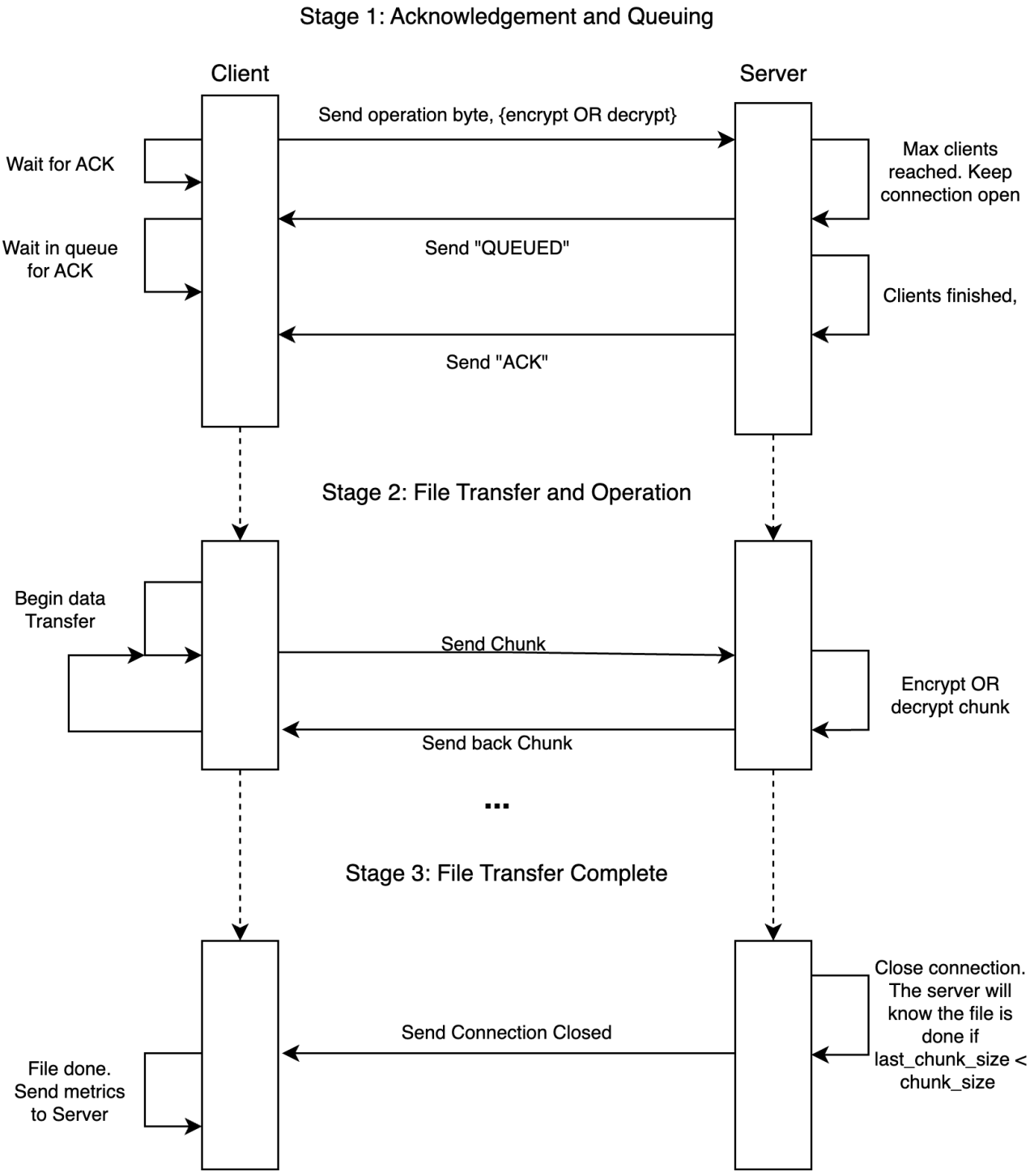
Each client connection operates on a stream of 1MB chunks to balance memory usage with transfer efficiency. The chunk size limit also allows for numerous clients to be simultaneously processed without exceeding memory usage. Figure 3.6 shows the client/server relationship.

The complete encryption process follows three stages: initial handshake with optional queuing, chunked file transfer with immediate processing feedback, and completion with metrics collection. Additionally, since all devices exist on the same physical link through an ethernet switch, several TCP options are tweaked to reduce overhead, since not as much TCP reliability mechanisms are necessary:

- `TCP_NODELAY` disables Nagle's algorithm, preventing transmission delays.

- `SO_SNDBUF/SO_RCVBUF` set to 4MB to match the 1MB chunk processing.
- `TCP_QUICKACK` forces immediate acknowledgment.
- `TCP_SLOW_START_AFTER_IDLE` disabled since bandwidth is consistent.
- Keepalive settings tuned for faster connection recovery (60s idle, 10s interval).

Figure 3.6: Full Server-Client Exchange Diagram



Test Orchestration

Now to create repeatable testing, we have implemented a test orchestration and metrics collection system. A python script, (tester.py) will be running on the master device (192.168.1.100) that manages test execution and data collection, while a receiving script (wait-for-commands.py), will be running on each Raspberry Pi in the cluster. The entire process comprises of three main components:

1. Test Coordination (tester.py):

   Executes four standardised test scenarios by distributing instructions to each client device on port 8080 in wait-for-command.py. These instructions specify the number of client threads and filesizes to run for each of the Raspberry Pis. Furthermore, unique SQLite databases are created per test run, where the server and client threads can post metrics. Lastly, the completion of each client thread is monitored through HTTP endpoints.

2. Client Device Management (wait-for-command.py):

   Each Raspberry Pi (192.168.1.11-14) runs two services:

   - Command server (same port as server-client, 8080, but different IP since it is wait-for-command.py running on the Raspberry Pis):
     - Receives test parameters (encrypt or decrypt, size of file to send).
     - If the 'test_files/' directory containing the test files does not exist, then the suite of test files will be generated with random text.
     - Then, spawns client processes.
   - Completion Server (port 8081):
     - Tracks the completion status of each client through polling them.
     - Notifies the orchestrator (tester.py), when a test is complete.

3. Metrics Collection (tester.py):

   Here, two types of schema are collected for server metrics and client metrics.

   The server metrics will be a continuous sampling at 100ms intervals, containing:

   - CPU/Memory usage.
   - XADC temperature readings.
   - Total bytes processed at sample time.

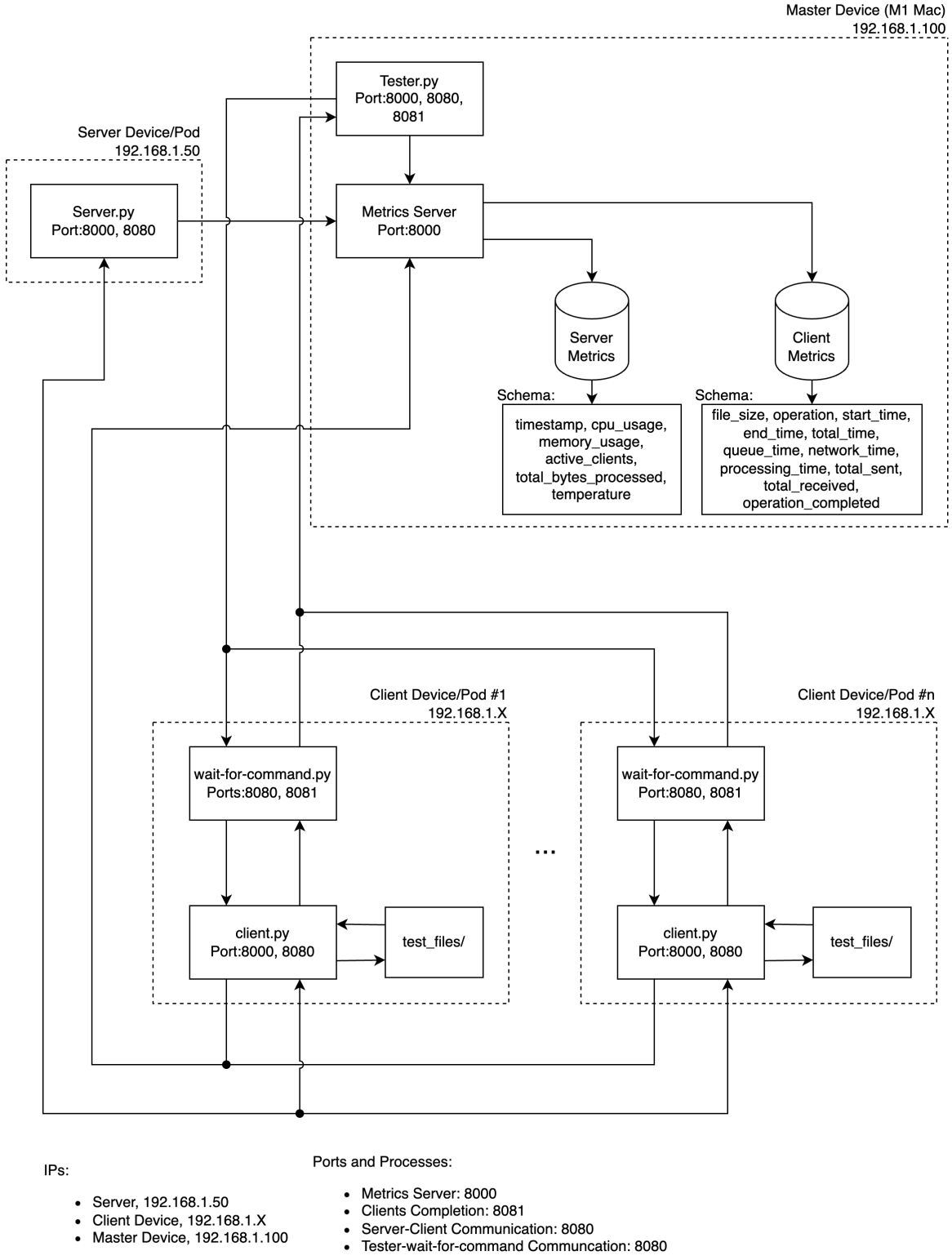
   The client metrics will be posted to the HTTP metrics server at the completion of the server/client exchange, this schema consists of:

   - The file size that was sent and the operation type (encryption or decryption).
   - A complete timing breakdown (time spent in queue, time spent on network transfers and time spent processing).

- A true/false flag indicating whether the operation was successful or not.

This comprehensive data collection provides a detailed performance comparison across different core configurations while maintaining consistent test conditions. A complete overview of the interactions between these processes can be seen in Figure 3.7.

Figure 3.7: Full Software Interactions Overview Diagram



Chapter 4

Evaluation

4.1 Raw Performance Benchmarks

Here, the different configurations of VexRiscv: single-core, dual-core and quad core; will be compared in terms of performance alongside the Raspberry Pi Model 4B 1GB. For each one, we will run the performance benchmark, *stress-ng*, which profiles the IO overhead and memory usage of multiple cores, along with *Iperf3*, to gauge ethernet throughput capabilities.

4.1.1 Iperf3

As ethernet is a vital component of the application, it makes sense to evaluate the capabilities of the link, especially the average bitrate we can expect. After running `iperf3 -s`, which sets the device as a server and listens, another Raspberry Pi was chosen from the cluster to act as a client via running:

```
iperf3 -c 192.168.1.50 -t 30 -i 1 -w 8K -P 1 -R
```

This begins a single-threaded (`-P 1`), client that sends and receives TCP transmissions to *192.168.1.50*, for 30 seconds, sampling the bitrate every second (`-i 1`). Most notably, it constrains the TCP window size (`-w 8K`), to 8Kb, which matches the current size of the board's TX or RX ethernet buffers, more closely resembling the stop-start transfers in our software setup. Here are the results:

Table 4.1: Ethernet Throughput Comparison of Configurations

	VexRiscvSMP, 100MHz			RPi
	Single	Dual	Quad	4B 1GB
Amount Transferred (MB)	31.5	35.5	42.0	1.13k
Amount Recieved (MB)	31.4	35.4	41.9	1.13k
Sending Bitrate (MBits/s)	8.78	9.90	11.7	323
Receiving Bitrate (MBits/s)	8.77	9.89	11.7	323
TCP Retransmissions	0	0	1	0

It is clear that the gigabit ethernet capabilities of the Raspberry Pi far outweigh the ethernet capabilities of the board, achieving 26x more throughput than that of the quad-core VexRiscvSMP.

Keep in mind as well, the Raspberry Pi is throttled because of the 8Kb window size we set. Additionally, the core amount does have an effect on the bitrate as the quad core is 1.8MBits/s faster than the dual core, which is 1.12MBits/s faster than the single core. However, an overall improvement of 32% across cores is basically negligible since the absolute performance is still far below what is required for what is essentially a high-speed ethernet application. It is clear from this benchmark alone that the bitrate will be a significant bottleneck for the design.

4.1.2 Stress NG

Stress-ng is a versatile benchmarking tool designed to stress test various components of a CPU. The command:

```
stress-ng --cpu $CORE_COUNT --io 2 --vm 1 --vm-bytes 128M --timeout 60s
--metrics-brief
```

Runs a set of tests in parallel. The rest of the arguments determine the amount of tests and the type: `--cpu`, creates CPU-intensive tasks equal to the core count; `--io`, creates two I/O-intensive tasks; and `--vm`, allocates and uses 128MB of virtual memory. This will evaluate for us how the system performs under combined CPU, I/O, and memory pressure, as well as how these metrics vary with the amount of cores. Additionally, the test will timeout after 60 seconds or until 6·`$CORE_COUNT` CPU bogo ops have been completed. Here are the results:

Table 4.2: Stress-ng Comparison of Configurations

	VexRiscvSMP, 100MHz			RPi
	Single	Dual	Quad	4B 1GB
CPU bogo ops	6	12	24	12,483
CPU real time (s)	125.34	119.09	121.30	60.03
CPU usr time (s)	78.32	164.00	378.16	146.85
CPU sys time (s)	0.01	0.12	0.09	0.03
CPU bogo ops/s (real time)	0.05	0.10	0.20	207.94
CPU bogo ops/s (usr+sys time)	0.08	0.07	0.06	84.99
IO bogo ops	21,794	31,190	27,819	440,066
IO real time (s)	60.00	60.01	60.00	60.00
IO usr time (s)	2.74	3.56	3.42	10.34
IO sys time (s)	25.71	44.44	66.44	48.56
IO bogo ops/s (real time)	363.22	519.76	463.65	7,334.31
IO bogo ops/s (usr+sys time)	766.05	649.79	398.21	7,472.11
VM bogo ops	2,280	3,053	2,464	617,668
VM real time (s)	61.92	62.54	61.54	60.16
VM usr time (s)	7.57	10.72	18.10	27.27
VM sys time (s)	7.60	14.93	18.32	6.78
VM bogo ops/s (real time)	36.82	48.82	40.04	10,266.29
VM bogo ops/s (usr+sys time)	150.30	119.03	67.66	18,143.58
Total time taken (s)	125.45	120.31	122.76	60.00

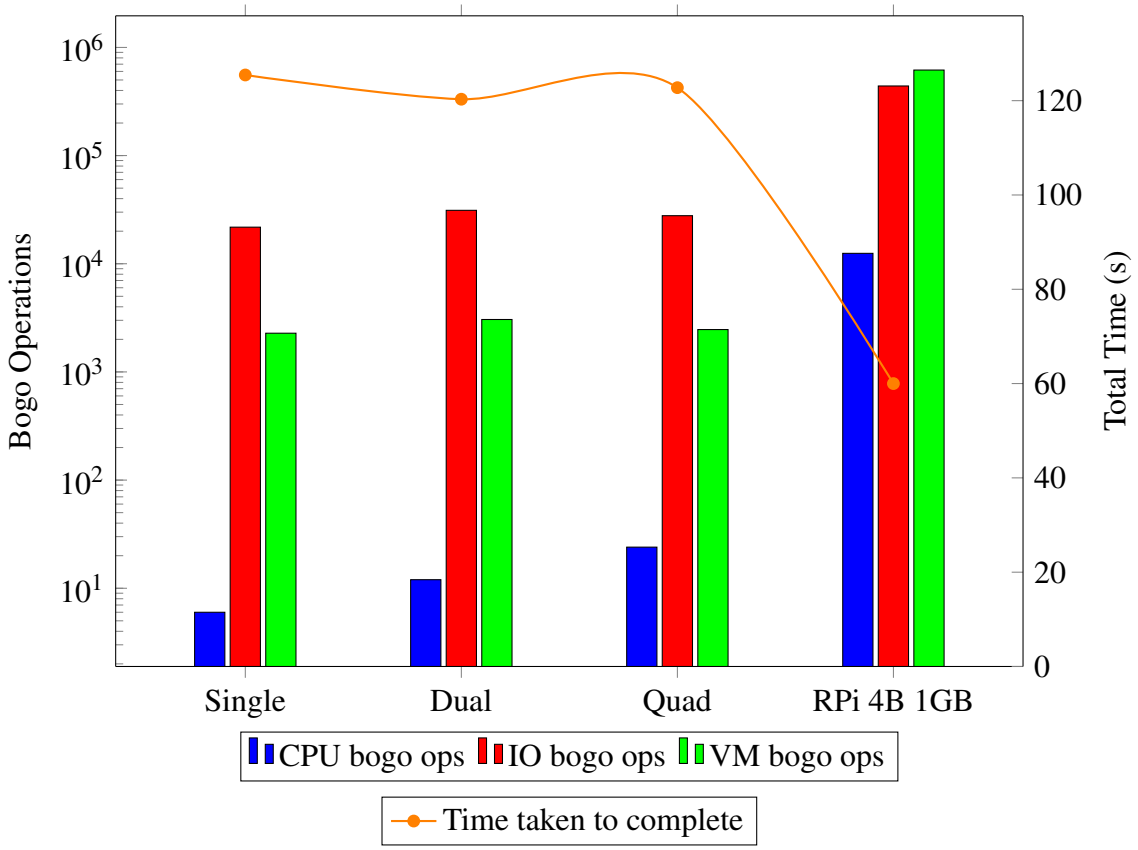


Figure 4.1: Stress-ng Performance Comparison Across Configurations

Immediately, we are presented with an extreme difference in performance scaling, which is to be expected when comparing an FPGA-based softcore processor to a Raspberry-Pi single-board computer. There is still nuance, however, between the different core configurations of the VexRiscvSMP. For instance, the amount of CPU bogo ops tend to scale linearly with the core count, maintaining a similar execution time (see ”CPU real-time” in table: 4.2). Other than this, there are no clear trends for the IO or VM bogo operations. It seems that once we extend to four cores, there are resource contentions with the IO bus and memory, leading to the quad core performing slightly less favourably than the dual core in these operations. Therefore, IO and memory have the potential to be another two bottlenecks, in addition to the low ethernet throughput, (although ethernet is still the dominant bottleneck). This can give us an estimation of how the dual core will perform relative to the quad core, *i.e.*, it may end up performing better since the quad core will not only have to contend with IO/memory constraints from the shared RAM module, but also the limited ethernet buffers.

Overall, out of the VexRiscvSMPs, dual core performed the best, but only if we exclude raw CPU power. As our application depends more on resource read/writes and management, dual core may prove to be the best configuration; besides the Raspberry Pi of course.

4.2 Results of Application & Analysis

In this section, we will now compare how each configuration performed running the exact same full suite of tests on our application defined in Section 3.2.2. We will then take the best performing out of the VexRiscv configurations and see if we can improve the design to achieve even more throughput.

4.2.1 Test Suite Full Outline

The tests will focus only on encryption, since decryption and encryption times are the same. We are aiming to deduce how each configuration performs with single or multiple clients. Therefore, the test outline is as such:

1. Basic test 1:
   1 client with 1 100mb file.

2. Basic test 2:
   10 clients with varying file sizes: 2mb, 5mb, 10mb, 20mb.

3. Basic test 3:
   20 clients with varying file sizes, in 10 client chunks.

4. Large scale test:
   100 clients with varying files sizes, in 10 client chunks.

While these tests are running, we are collecting a multitude of metrics regarding overall performance, but the primary result we are concerned with is the total bytes processed (or bytes encrypted) per second. This single metric is all we need to determine which amount of cores is superior for the application. It is a standard average, calculated from:

$$Average\ Bytes\ Processed(Bytes/s) = \frac{Total\ Bytes\ Processed}{Test\ Duration}$$

Where *'Total Bytes Processed'* is the total amount of bytes processed during the test, and *'Test Duration'*, is how long the test took to execute. This is calculated from the server-side, as all the information is present there.

On the client-side, we are also keeping track of the client's total time for the transaction, time spent in the queue, time spent on network transfers and the time spent processing on the server, *i.e.*encrypting. Total time, network time and queuing time are collected in real time on client-side, leaving the processing time as the remainder, calculated via:

$$Processing\ Time(s) = Total\ Time - Network\ Time - Queue\ Time$$

Lastly, all timings were collected using the *'Time'*, Python standard library.

4.2.2 Single Core Results

Single Core Tests	Basic 1	Basic 2	Basic 3	Large Scale 100
Total Bytes Processed (Mb)	104.86	84.93	169.87	849.35
Avg Bytes per Second (Kb/s)	221.74	183.57	187.14	184.78
Test Duration (s)	472.89	462.69	907.70	4596.65
Total Network Time (s)	173.70	188.40	377.81	2100.95
Total Processing Time (s)	299.18	274.29	529.89	2495.71
Avg Queue Time (s)	0.15	8.28	7.66	8.05

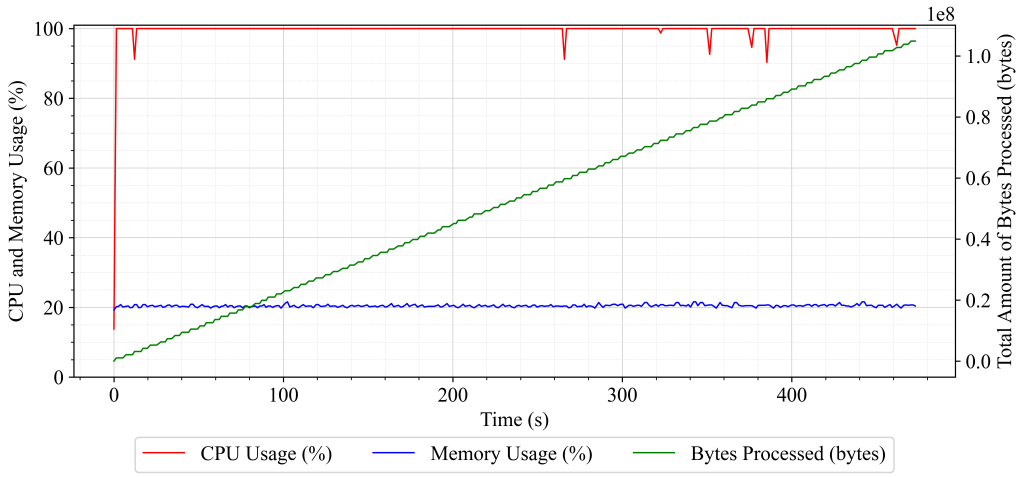
Table 4.3: Single Core Configuration Resulting Metrics

The single core VexRiscvSMP configuration at 100MHz establishes our baseline performance characteristics. During Basic Test 1 (single client, 100MB file), the configuration achieved its peak throughput of 221.74 KB/s with minimal queuing overhead (0.15s).

However, we begin to see performance degradation in multi-client scenarios. Throughout Basic Tests 2 and 3, average throughput drops to approximately 185KB/s as the system handles concurrent client requests. The impact of single-threaded execution is reflected in the increasing queue times, jumping from 0.15s to 8.28s with just 10 clients. The Large Scale test with 100 clients demonstrates the configuration's scaling limitations, maintaining the same constrained throughput of 184.78 KB/s while requiring 2100.95s for network transfers and 2495.71s for processing (totalling to 4596.65s or 1hr 16mins).

Figure 4.2 reveals a critical performance aspect - CPU utilisation remained at 100% throughout all the tests, with memory usage stable at around 20%. This continuous maximum utilization indicates that the core is constantly engaged in either encryption or network handling tasks, with no idle periods for additional workloads. See Appendix A.1 to A.4 for full results for the single core configuration.

Figure 4.2: Single Core, Basic Test 1 Server Metrics



Throughout all tests, the single core configuration maintained an operating temperature between 65-66°C, indicating that despite the constant 100% CPU utilization, the design operates well within the Artix-7's thermal constraints (0-85°C).

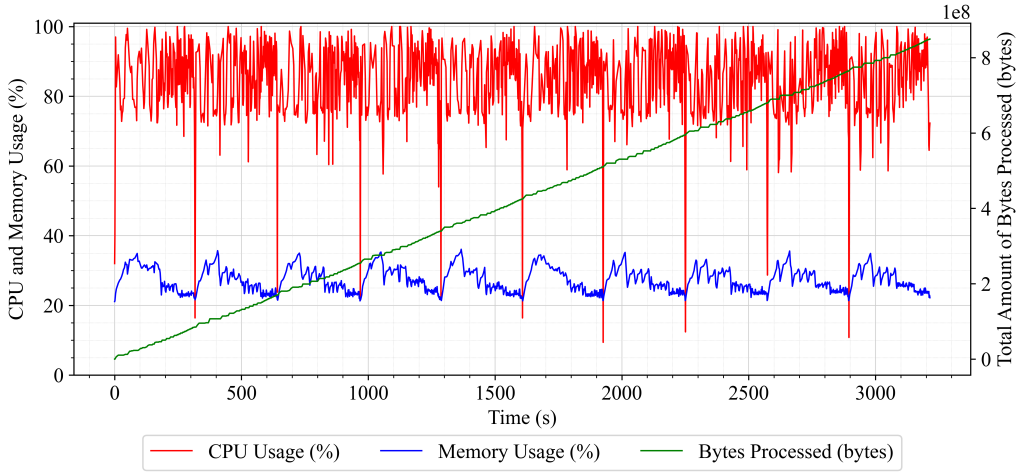
4.2.3 Dual Core Results

Dual Core Tests	Basic 1	Basic 2	Basic 3	Large Scale 100
Total Bytes Processed (Mb)	104.86	84.93	169.87	849.35
Avg Bytes per Second (Kb/s)	272.97	265.26	272.79	264.22
Test Duration (s)	384.13	320.19	622.70	3214.54
Total Network Time (s)	152.47	110.49	256.35	1166.14
Total Processing Time (s)	231.66	209.70	366.36	2048.41
Avg Queue Time (s)	0.07	4.35	3.94	4.42

Table 4.4: Dual Core Configuration Resulting Metrics

The dual core configuration demonstrates substantial improvements over the single core design. Running at 100MHz, throughput increases by approximately 45% to a consistent 264-272 KB/s across all tests. Figure 4.3, shows a distinctive CPU utilisation pattern oscillating between 75-100%, contrasting with the single core's constant 100% usage, while memory consumption remains similar at 25-30%.

Figure 4.3: Dual Core, Large Scale, 100 clients, Server Metrics



The parallel processing capability significantly reduces processing bottlenecks. Average queue times are halved from 8.28s to 4.35s in Basic Test 2, and the Large Scale test completes in 3214.54s (53min) compared to the single core's 1hr 16min. Network transfer times also improve considerably, with the Large Scale test requiring only 1166.14s, versus 2100.95s on the single core, suggesting better management of concurrent network and encryption operations.

Temperature measurements show minimal change from the single core, stabilising at 66°C during operation. This thermal stability, combined with the improved performance metrics, indicates that a more efficient workload distribution between the cores was achieved. For full results and plots, see Appendix A.5 to A.8.

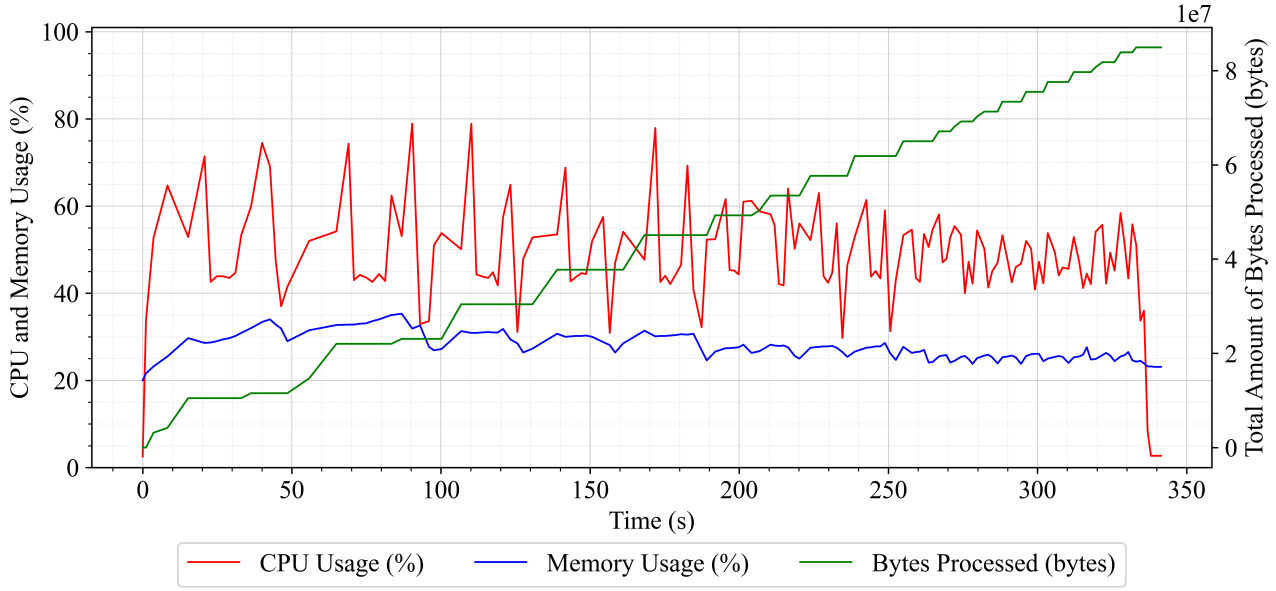
4.2.4 Quad Core Results

Quad Core Tests	Basic 1	Basic 2	Basic 3	Large Scale 100
Total Bytes Processed (Mb)	104.86	84.93	169.87	849.35
Avg Bytes per Second (Kb/s)	269.92	248.77	251.97	245.57
Test Duration (s)	388.48	341.42	674.16	3458.68
Total Network Time (s)	171.60	100.68	209.94	902.06
Total Processing Time (s)	216.88	240.74	464.23	2556.62
Avg Queue Time (s)	0.08	4.37	3.65	3.83

Table 4.5: Quad Core Configuration Resulting Metrics

The quad core configuration reveals the limitations of adding additional cores to a bandwidth-constrained system. Despite doubling the processing resources from the dual core design, CPU utilization never exceeds 80% as shown in Figure 4.3, indicating that cores are frequently idle waiting for data. This resource starvation prevents the throughput from improving compared to the dual core (245-269KB/s compared to 264-272KB/s).

Figure 4.4: Quad Core, Basic Test 2 Server Metrics



This bottleneck becomes more apparent in the Large Scale test, which takes 3458.68s (57mins), to complete versus the dual core's 53mins. The increased processing time (2556.62s vs 2048.41s) suggests that the single ethernet interface and shared memory bus cannot funnel data to the cores fast enough to utilise the additional processing capacity. Queue times remain similar to the dual core at around 4.37s, indicating that the bottleneck lies in data transport rather than processing capability.

Temperature remains stable at 66°C, matching the dual core configuration. This thermal consistency aligns with the lower CPU utilisation, as the cores spend significant time idle waiting for data. The full results can be seen in Appendices A.9 to A.12.

4.2.5 Raspberry Pi Results

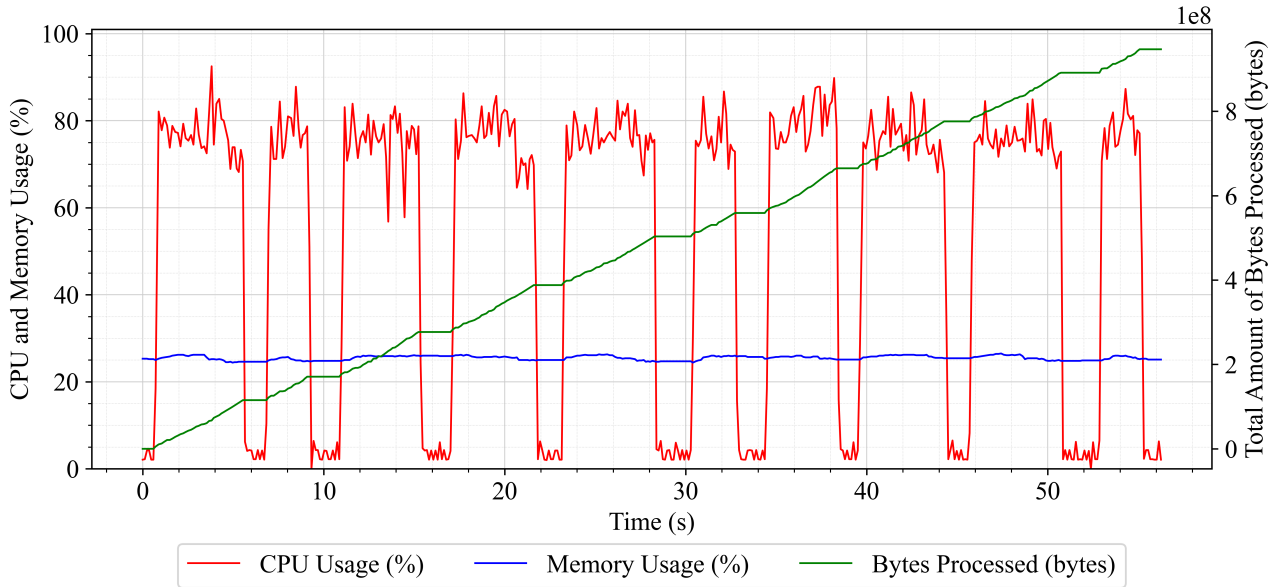
Raspberry Pi Tests	Basic 1	Basic 2	Basic 3	Large Scale 100
Total Bytes Processed (Mb)	104.86	55.57	217.06	946.86
Avg Bytes per Second (Kb/s)	8042.64	13853.24	17895.55	16829.84
Test Duration (s)	13.04	4.01	12.13	56.26
Total Network Time (s)	8.06	2.05	8.13	37.50
Total Processing Time (s)	4.97	1.96	4.00	18.76
Avg Queue Time (s)	0.00	0.03	0.02	0.03

Table 4.6: Raspberry Pi Resulting Metrics

The Raspberry Pi 4B 1GB, functioning as our hardware performance baseline, demonstrates dramatically superior performance across all metrics. With its physical cores running at 1.8GHz and a full Linux networking stack (not via BusyBox), throughput ranges from 8-17 MB/s - approximately 40 times faster than any VexRiscvSMP configuration. Even under high multi-client loads, queue times remain negligible at 0.03s compared to the several seconds seen in the FPGA implementations.

The efficiency of the hardware architecture is evident in the processing statistics. The Large Scale test completes in just 56.26s versus the thousands of seconds required by VexRiscvSMP configurations. CPU utilisation, best illustrated by Figure 4.5, follows an efficient pattern, cycling between active processing and idle states rather than maintaining constant high utilisation. Memory usage stabilises at 25-30%, similar to the FPGA implementations despite the significantly higher throughput.

Figure 4.5: Raspberry Pi, Large Scale, 100 clients, Server Metrics



Operating temperatures remain well-controlled between 40-44°C during the Large Scale test, aided by the cooling fan and the ARM core's power management capabilities. While not approaching the theoretical limits of its Gigabit ethernet interface, the combination of high clock rate, physical CPU cores, and integrated peripherals demonstrates the current performance gap between FPGA-based

softcores and physical processor implementations. Full results can be found in Appendices A.13 to A.16.

4.3 Improved Dual Core Design

Based on our analysis, the dual core design emerged as the optimal candidate for improvement. Unlike the quad core which suffered from resource starvation, the dual core demonstrated efficient resource utilisation while maintaining consistent performance scaling. The fewer cores also left additional FPGA resources available for optimisations.

Two key improvements were implemented:

1. Clock Frequency Increase:
    - Base frequency raised from 100MHz to 150MHz (+50%)
    - Maximum stable frequency with slight timing violations.
    - Provides proportional improvement to processing capability.
2. Ethernet Buffer Expansion: The remaining FPGA resources (about 35%), were then dedicate to larger ethernet buffers. We managed to quadruple the RX/TX buffers from 0x1000 to 0x8000 bytes each (8KB to 32KB).

4.3.1 New Raw Benchmark Performance Results

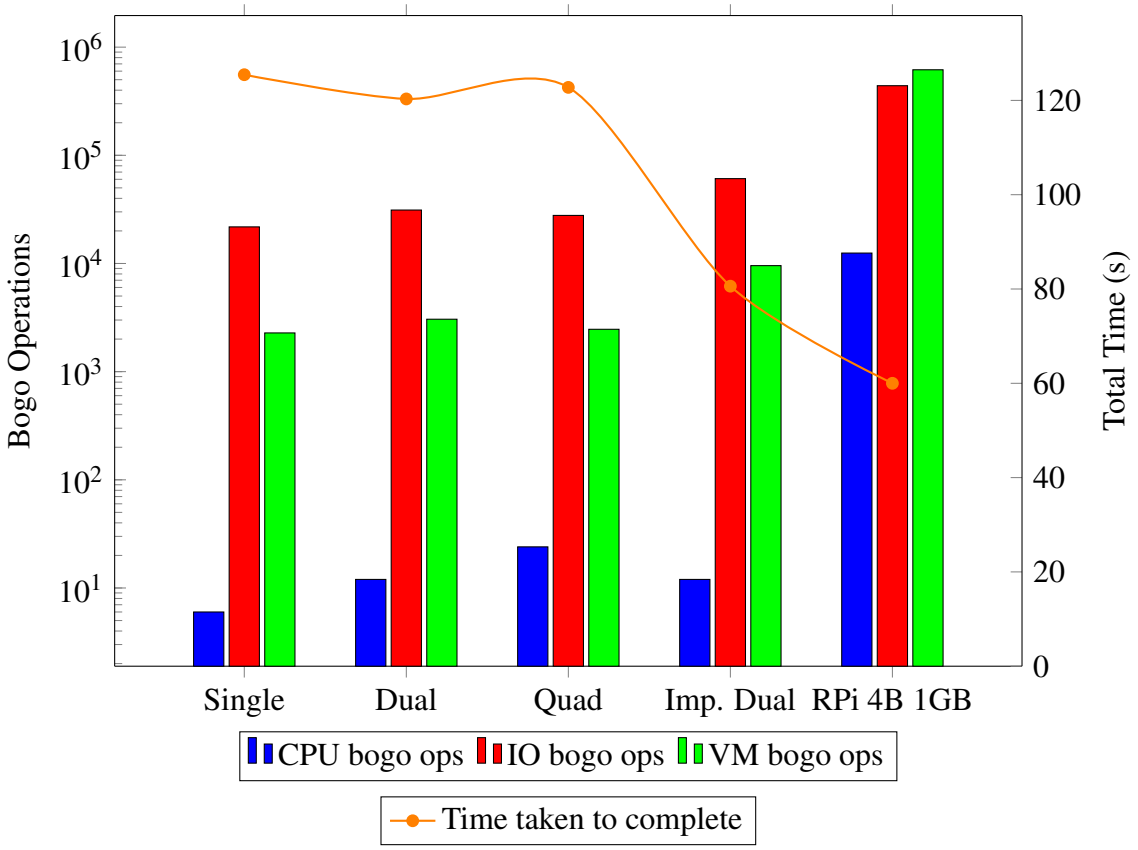
The impact of these improvements is immediately evident in the new raw benchmark results. IPerf3 testing shows network throughput increasing from 9.89 Mbits/s to 21.6 Mbits/s, while maintaining zero packet retransmissions (see Table 4.7).

Stress-ng results demonstrate significant gains in I/O operations (60,868 vs 31,190) and virtual memory operations (9,552 vs 3,053), indicating a superior resource utilisation (See Figure 4.6).

	VexRiscvSMP, 100MHz			150MHz	RPi
	Single	Dual	Quad	Dual	4B 1GB
Amount Transferred (MB)	31.5	35.5	42.0	77.4	1.13k
Amount Recieved (MB)	31.4	35.4	41.9	77.2	1.13k
Sending Bitrate (MBits/s)	8.78	9.90	11.7	21.6	323
Receiving Bitrate (MBits/s)	8.77	9.89	11.7	21.6	323
TCP Retransmissions	0	0	1	0	0

Table 4.7: New Ethernet Throughput Comparison of Configurations

Figure 4.6: Stress-ng Performance Comparison Across All Configurations



4.3.2 Results and Analysis

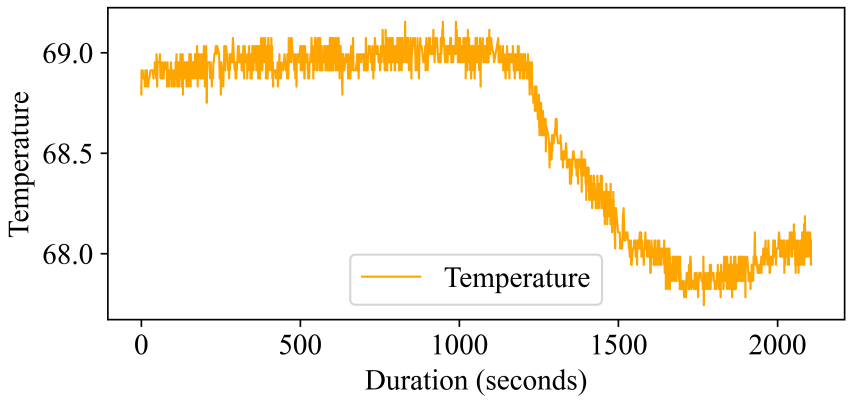
Improved Dual Core Tests	Basic 1	Basic 2	Basic 3	Large Scale 100
Total Bytes Processed (Mb)	104.86	84.93	169.87	849.35
Avg Bytes per Second (Kb/s)	376.04	406.06	393.59	403.36
Test Duration (s)	278.85	209.17	431.59	2105.67
Total Network Time (s)	51.08	71.65	108.20	646.95
Total Processing Time (s)	227.77	137.52	323.39	1458.72
Avg Queue Time (s)	0.08	3.11	1.95	2.81

Table 4.8: Dual Core Improved, Basic Test 2 Server Metrics

These optimisations translate directly to improved encryption performance. The improved design achieves throughput between 376-406 KB/s, approximately 50% higher than the original dual core configuration. Processing times show similar improvement, with the Large Scale test completing in 2105.67s versus 3214.54s. Queue times reduce from 4.35s to 3.11s on average, suggesting better overall system responsiveness.

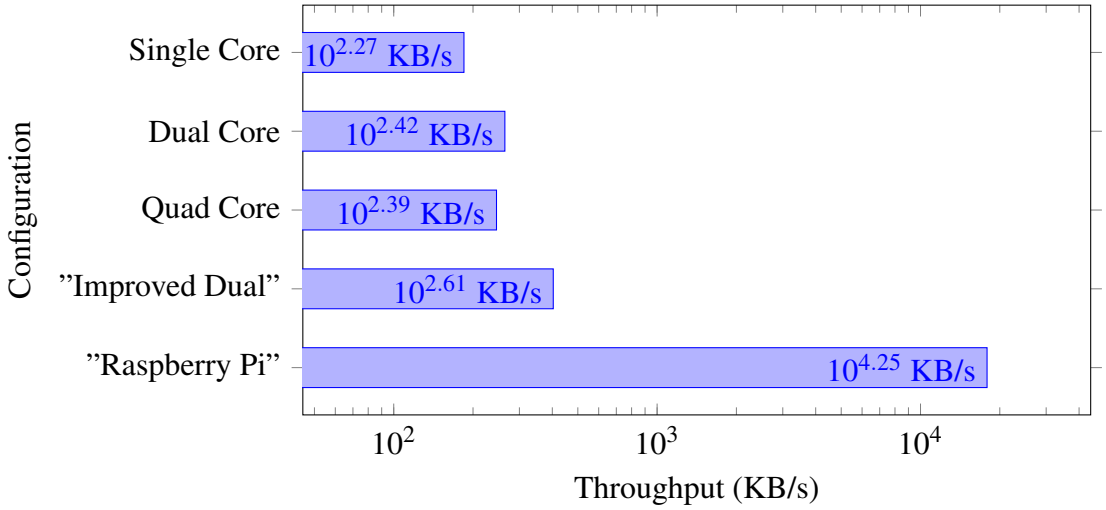
Operating temperature stabilizes slightly higher at 68-69°C, a modest increase that reflects the higher clock rate while remaining well within the Artix-7's thermal specifications. Interestingly, in the temperature plot for the improved dual core (Figure 4.7), once the air-conditioning in the lab was turned on, there was a gradual drop in temperature of one degree which was observed.

Figure 4.7: Raspberry Pi, Large Scale, 100 clients, Server Metrics



4.4 Application Results Summary

Figure 4.8: Large Scale 100 Client Test Encryption Throughput Across Configurations



The performance analysis across configurations reveals clear trends in the viability of FPGA-based softcore processors for network security applications. Single core performance established a baseline of 221 KB/s maximum throughput, with dual core showing a 23% improvement to 272 KB/s through effective parallel processing. The quad core configuration, however, demonstrated diminishing returns due to resource starvation, achieving only 269 KB/s despite doubled processing resources. The improved dual core design, implementing a 150MHz clock rate and quadrupled ethernet buffers, achieved the best FPGA performance at 406 KB/s - an 84% improvement over the baseline.

However, these improvements still fall significantly short of the Raspberry Pi's 17,895 KB/s throughput, highlighting the performance gap between FPGA-based softcores and physical processor implementations. These results suggest that while FPGA softcore implementations can provide functional network security solutions, their performance remains fundamentally constrained by architectural and physical limitations.

4.5 Benchmarking TLS

We will now observe how much overhead is introduced by running the full TLS1.3 stack on the SCPNS. We will begin by generating the key and the certificates using a 2048-bit RSA:

```
openssl req -x509 -newkey rsa:2048 -keyout key.pem -out cert.pem -days 365
-nodes
```

This process took roughly two minutes, whereas on hardcore processors, such as the Raspberry Pi, it is instant. Next, we need to start a server on the SCPNS running TLS1.3. Forunately, OpenSSL also has testing utilities for this scenario. We can start a server using our newly generated key and certificate:

```
openssl s_server -cert cert.pem -key key.pem -accept 8443 -cipher
AES256-GCM-SHA256
```

From a client device, such as the Raspberry Pi, we can also use OpenSSL to act as a client and begin a TLS1.3 end-to-end encrypted stream via:

```
dd if=/dev/zero bs=1K count=64 2>/dev/null | \
    openssl s_client -connect 192:168.1.50:8443 \
    -cipher AES256-GCM-SHA256 2>/dev/null | \
    dd of=/dev/null 2>/dev/null
```

The surrounding ‘dd’ commands specify the size of the message to send, in this case it is 64KB. To get a comprehensive overview, we will run streams of sizes: 64KB, 128KB, 256KB, 512KB and 1024KB; each three times to get an average throughput. Table 4.9 shows the results.

Stream Size (KB)	**(No HW) AES (KB/s)**	**(HW) AES (KB/s)**	**Difference**
64	20.44	19.22	-5.97%
128	12.76	13.24	+3.76%
256	12.80	12.80	0%
512	14.90	13.24	-11.14%
1024	13.70	13.75	+0.36%

Table 4.9: TLS1.3 Benchmark on the SCPNS, AES256-GCM-SHA256 Cipher, Results

Disappointingly, these benchmarks reveal that the hardware acceleration of AES operations on the SCPNS had minimal impact on overall throughput. Across the different stream sizes the performance differences between hardware and software implementations ranged from -11.14% to +3.76%, with some sizes showing negligible change. Despite the 4x improvement seen in raw AES benchmarks, these gains did not translate to TLS performance, suggesting that the bottleneck lies elsewhere in the protocol stack - likely in the TLS overhead (handshake, key exchange, *etc.*), network interface, or memory subsystem rather than in the cryptographic operations themselves. Observe as well, that our raw TCP throughput for the improved dual core was 21.6Mbits/s (Table 4.7). By running TLS1.3, the throughput is significantly reduced to 12-20KB/s.

4.6 Utilisation of Configurations

Analysis of FPGA resource utilisation across configurations reveals how the design scales with additional cores and improvements:

Logic Resources (LUTs and FFs):

- Single core uses 41.24% of available LUTs and 19.05% of FFs.
- Dual core increases to 60.19% LUTs and 27.25% FFs.
- Quad core reaches 94.13% LUTs and 40.73% FFs.
- Improved dual core uses 89.81% LUTs and 28.45% FFs.

The near-linear scaling in register usage (roughly +8% FFs per core) suggests efficient core replication. However, LUT utilisation shows diminishing returns, with the quad-core configuration nearly maxing out the device's logic resources. This solidifies why we could not further optimise the quad-core configuration - there weren't any resources left.

Memory Resources:

- BRAM utilisation scales from 74% (single) to 83% (dual) to 99% (quad).
- The improved dual core uses 100% of available BRAM, mainly due to enlarged buffers.
- This indicates memory resources become a limiting factor for further optimisation.

DSP Usage:

- DSP utilisation scales linearly with core count (4, 8, and 16 DSPs respectively).
- Only using up to 17.78% of available DSP resources.
- DSPs are primarily used for address generation and memory management.

The improved dual-core design, while using similar FF resources to the standard dual-core, shows significantly higher LUT and BRAM usage due to its increased amount of buffers for ethernet and interconnects. This represents a better balance of resource utilisation versus performance, as demonstrated by the throughput results (Table 4.7). The full utilisation of the improved dual core can be seen in table 4.10.

Resource Type	Used	Available	Utilisation
Slice LUTs	18,680	20,800	89.81%
- LUT as Logic	15,124	20,800	72.71%
- LUT as Memory	3,556	9,600	37.04%
Slice Registers (FFs)	11,836	41,600	28.45%
Block RAM Tiles	50	50	100.00%
- RAMB36	42	50	84.00%
- RAMB18	16	100	16.00%
DSPs	8	90	8.89%

Table 4.10: Improved Dual Core Utilisation

4.7 Timing Analysis

The timing analysis reveals critical path constraints that limit our system's maximum operating frequency. While all configurations are stable at 100MHz, attempts to increase the clock rate to 150MHz expose several timing violations:

Critical Paths:

Our most significant timing violations occur in the DDR3 memory interface, with:

- Worst Negative Slack (WNS): -2.805ns
- Total Negative Slack (TNS): -9230.256ns
- 11,401 failing endpoints out of 57,626 total endpoints

Figure 4.9, shows the majority of these failed paths are attached to the '`main_crg_clkout0`' clock domain.

Name	Slack ^1	Levels	High Fanout	From	To
Path 55	-2.805	10	13	main_soclinux_sdram_bankmachine7_pipe_valid_source_payload_addr_reg[9]/C	main_soclinux_sdram_bank...ine5_trascon_ready_reg/R
Path 56	-2.805	10	13	main_soclinux_sdram_bankmachine7_pipe_valid_source_payload_addr_reg[9]/C	main_soclinux_sdram_bankm...hine5_trccon_ready_reg/R
Path 57	-2.764	8	176	tag_mem_adr0_reg_rep[0]/C	data_mem_grain7_reg_448_511_0_2/RAMA/WE
Path 58	-2.764	8	176	tag_mem_adr0_reg_rep[0]/C	data_mem_grain7_reg_448_511_0_2/RAMB/WE
Path 59	-2.764	8	176	tag_mem_adr0_reg_rep[0]/C	data_mem_grain7_reg_448_511_0_2/RAMC/WE
Path 60	-2.764	8	176	tag_mem_adr0_reg_rep[0]/C	data_mem_grain7_reg_448_511_0_2/RAMD/WE
Path 61	-2.731	9	176	tag_mem_adr0_reg_rep[0]/C	data_mem_grain1_reg_448_511_3_5/RAMA/WE
Path 62	-2.731	9	176	tag_mem_adr0_reg_rep[0]/C	data_mem_grain1_reg_448_511_3_5/RAMB/WE
Path 63	-2.731	9	176	tag_mem_adr0_reg_rep[0]/C	data_mem_grain1_reg_448_511_3_5/RAMC/WE
Path 64	-2.731	9	176	tag_mem_adr0_reg_rep[0]/C	data_mem_grain1_reg_448_511_3_5/RAMD/WE

Figure 4.9: Critical Path Failures W.R.T `main_crg_clkout0`

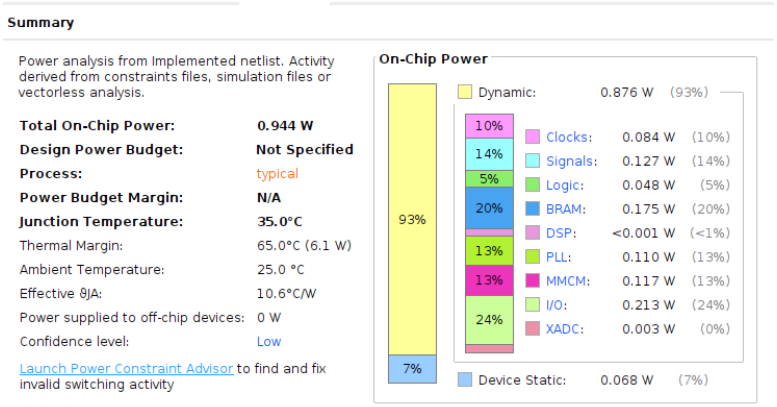
These violations center around in the DDR3 controller interface where tight timing requirements exist for address and control signals. The negative slack of -2.805ns indicates we're missing our timing requirements by approximately 3 nanoseconds on the worst paths. However, despite these failed paths, we were still able to complete the full test suite at 150MHz. Presumably, there must be a sufficient error margin in the DDR3 protocol to accommodate this timing uncertainty.

This analysis suggests that to achieve a higher clock rate on the Digilent Arty A7, the memory controller path from the VexRiscvSMP should be optimised first, however, this is out of scope for this thesis.

4.8 Power Usage Analysis

Power consumption was analysed both theoretically through Vivado's power analysis and through real-world measurements using the board's XADC (see XADC in Digilent Reference Manual [9]).

Figure 4.10: Synthesis With AES Instructions Power Report.

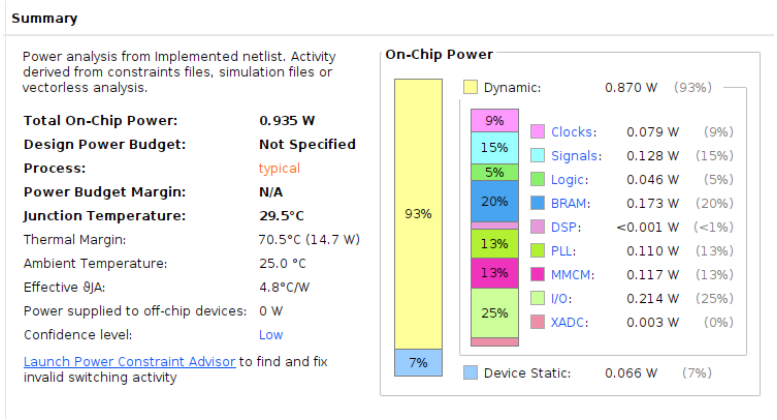


Core power consumption was evaluated both with and without custom AES instructions:

- Without AES instructions: 0.935W.
- With AES instructions: 0.944W.
- Power overhead of AES: ∼9mW (0.96% increase).

The AES-enabled configuration power is distributed across:

Figure 4.11: Synthesis Without AES instructions Power Report.



- Dynamic Power: 0.876W (93%).
- I/O: 0.213W (24%).
- BRAM: 0.175W (20%).
- Signals: 0.127W (14%).
- Clocks: 0.084W (10%).
- Other (PLLs, MMCM, *etc.*): ∼0.277W (25%).
- Static Power: 0.068W (7%).

The PLL and MMCM can be disabled if power needs to be reduced more, as we do not use these components in the application. Furthermore, the total power used by the FPGA core was confirmed by reading the `VCCINT` register on the XADC.

Real-world measurements through the XADC's `VCCAUX` register show a total board power consumption of 1.77W. This higher figure accounts for all board components, FPGA core (0.944W), DDR3 memory module, Ethernet PHY, Voltage regulators and clock generators as well as the LEDs.

The total power consumption of 1.77W, while modest for a full-featured network security processor, is still significant for battery-powered or ultra-low-power applications. The FPGA maintains this power draw even when idle, as the design lacks power-saving states or clock gating mechanisms. Typically, this is solved by implementing a low power co-microcontroller to manage FPGA power states, *i.e.*power gating circuitry to completely disable the FPGA when not needed and a boot management system to quickly restore the FPGA state when encryption services are required.

Chapter 5

Conclusion

5.1 Summary

This project explored the viability of implementing a network security processor using RISC-V softcores on FPGA hardware, with a specific focus on AES encryption acceleration. Through the development of custom AES instructions and implementation of various VexRiscvSMP configurations on the Artix-7 FPGA, we demonstrated both the potential and limitations of FPGA-based security solutions. The optimal configuration—an improved dual-core design running at 150MHz with expanded ethernet buffers—achieved a maximum throughput of 406 KB/s, representing an 84% improvement over the baseline single-core implementation.

However, comparative analysis with a Raspberry Pi 4B revealed significant performance disparities, with the ARM-based system achieving throughput rates approximately 40 times higher than our best FPGA implementation. This gap widened further when implementing full TLS 1.3 encryption, where our hardware-accelerated AES instructions showed minimal impact on overall system performance, suggesting that the primary bottlenecks lie not in the cryptographic operations themselves but in the surrounding infrastructure—particularly ethernet throughput and memory access patterns. Despite these limitations, the project successfully demonstrated that FPGA-based security processors are feasible, albeit with performance characteristics better suited to low-to-medium throughput applications rather than high-performance computing scenarios.

5.2 Limitations and Security Vulnerabilities

The implementation faces several significant technical limitations and potential security concerns that should be considered:

Technical Limitations:

- Network Throughput: The most severe limitation is the 100Mbps ethernet PHY combined with limited buffer sizes, creating a fundamental bottleneck that persists even with increased processing power. Even with optimisations, the system achieves only 21.6 Mbps throughput on a physical switch, far below modern networking requirements.
- Memory Constraints: The shared memory architecture creates contention between cores, particularly evident in the quad-core configuration where additional processing power provided diminishing returns due to memory access bottlenecks.
- Clock Speed Ceiling: While we achieved 150MHz operation, timing violations in the DDR3 interface suggest we've reached the practical frequency limit for this design on the Digilent Arty A7. Higher frequencies would require significant redesign of memory interfaces.
- Power Management: The lack of power-saving states means the FPGA maintains constant power draw even when idle, limiting its suitability for battery-powered applications.

Security Vulnerabilities:

To reiterate, producing a fully-secure IoT device was out of scope for this thesis. However, to improve the security of this particular device, several areas must be considered:

- Side-Channel Attack Vulnerability: The fixed-frequency operation and consistent power consumption patterns could make the system vulnerable to power analysis attacks, as no countermeasures were implemented against side-channel analysis.
- Key Management: The current implementation uses hardcoded test keys and IVs for AES operations. A production system would require secure key storage and management capabilities, possibly through a dedicated secure element.
- Memory Protection: While RISC-V provides memory protection features through PMP, our implementation focuses solely on encryption performance and doesn't implement these security features, leaving potential vulnerabilities in memory access control. We also did not test or evaluate these features.
- TLS Implementation: The minimal performance improvement in TLS operations despite AES acceleration suggests potential vulnerabilities in how secure connections are handled, particularly in terms of session management and key exchange operations.

These limitations and vulnerabilities must be considered for future iterations of the design, particularly if the design is intended for production use in security-critical applications.

5.3 Future Improvements

While this project demonstrates the feasibility of an FPGA-based network security processor, the performance analysis revealed several critical bottlenecks and limitations that should be addressed in future iterations. The most significant improvements should target three key areas: the network infrastructure that limits overall throughput, the memory architecture that constrains multi-core scaling, and the TLS implementation that currently shows minimal benefit from hardware acceleration. Additionally, optimisations in power management and software integration could enhance the system's practicality for real-world deployment. The following sections detail specific improvements in each area, with particular attention to maintaining security while achieving better performance.

Network Infrastructure:

- Use an evaluation board with a gigabit ethernet PHY with direct memory access (DMA) to bypass CPU involvement in data transfers.
- Develop custom network offload engines to handle TCP/IP stack processing in hardware.
- Design more dedicated instructions for superior simultaneous network I/O and encryption operations (such as a TCP checksum acceleration).
- Consider an evaluation board with multiple ethernet interfaces, since we cannot just increase the clockrate of the softcore processor (parallelisation).

Memory Architecture:

In our timing analysis (Section 4.7), we determined that critical path failures begin at 150MHz around the memory interfaces with the DDR3 RAM module. Fixing these timing constraints would require some advanced knowledge of CPU architecture and interfacing, but there may be some tweaks available for LiteDRAM to try to mitigate these issues.

TLS Acceleration:

We determined in Section 4.5, that our raw TCP throughput becomes severely crippled due to the overhead of the full TLS1.3 stack. Improvements for this aspect of the design should be inspired by the research by Isobe *et al.* [23], by considering hardware acceleration for the more critical bottlenecks of the TLS stack. This should include:

- Implement hardware acceleration for key TLS 1.3 primitives:
  - RSA/ECC for asymmetric cryptography during handshakes

  - SHA-256/SHA-384 for hash functions
  - ChaCha20-Poly1305 as an alternative to AES-GCM
- Design a dedicated TLS protocol processor to offload handshake operations in an SoC.
- For TLS testing and implementation, there do exist alternatives better suited for embedded Linux than our OpenSSL test implementation. Two examples are wolfSSL and MbedTLS.

The python application currently uses an entire OpenSSL subprocess for encryption which can be performance intensive. This was necessary because the patch for the custom AES instructions was targeted for the LibreSSL library, and the python crypto library uses the default OpenSSL library.

Power Optimisation

We covered considerations for optimising the power usage when we conducted the power analysis in Section 4.8. To reiterate, these considerations should be:

- Implement clock gating for idle cores.
- Design power domains that can be completely disabled when not in use.
- Add a low-power microcontroller for power management and quick-wake capabilities.

Fortunately, the AES instructions only used a minimal amount of additional power and FPGA utilisation. This means that implementing more components of the TLS stack, such as the ones mentioned before, should also be cheap in terms of resources.

Software Stack

We do not have to rely on just hardware optimisations however, there is still significant room for improvement around the software layer. Two key considerations include:

- Replacing the Python-based server with a C implementation (using the POSIX API) for reduced overhead.
- Implement direct OpenSSL engine support rather than using subprocess calls, via crypto libraries for C or Python.

FPGA Platform

Lastly, choice of FPGA platform and evaluation board could provide some benefit, but the bottlenecks ultimately lie in the softcore processor itself. The simplest route to significantly improving the design would be to use modern FPGA families with hardened processor systems (*e.g.*, the Zynq line for ARM or Microchip's BeagleV-Fire for RISC-V), to combine the benefits of both hardware and software processing. Of course, however, this neglects the usage of a softcore processor which was not the aim of this thesis.

5.4 Issues During Development Resulting in Scope Adjustments

Several significant technical challenges encountered during development necessitated adjustments to the project's scope. These issues provide valuable insights for future similar implementations.

5.4.1 Major Issues

Zephyr Locking up on VexrsicvSMP

Despite following established patterns from similar RISC-V implementations like the Polarfire and SiFive boards, the Zephyr RTOS failed to properly initialise on our VexriscvSMP configuration. GDB debugging revealed critical errors due to spurious interrupts (see Figure 5.1), suggesting incompatibilities between our PLIC/CLINT implementation and Zephyr's expectations. While the interrupt controller configuration appeared correct in both device tree and defconfig files, the persistent crashes led to abandoning Zephyr in favor of Buildroot Linux, which without any additional effort, was able to correctly interpret our interrupt configuration (can be seen in Appendix A.8), thus providing a more stable platform for our application.

Socat Not Working on Buildroot Linux

Initial plans to use *Socat* (a more robust re-implementation of *Netcat*), for managing multiple client connections were thwarted by consistent `FD_SETSIZE` errors in Buildroot Linux:

```
root@buildroot:~# socat - TCP-LISTEN:1024,fork,reuseaddr
*** bit out of range 0 - FD_SETSIZE on fd_set ***: terminated
```

This limitation forced a pivot to a Python-based implementation. While this added development complexity, it ultimately proved beneficial by enabling easier integration of metrics collection through Python's standard libraries.

Not Enough RAM on Pi Model 4Bs for Docker Containers

The original test architecture planned to use Kubernetes (*microk*8*s* implementation for Pi Clusters), for client orchestration using minimal Debian images (150MB). However, the transition to Python-based clients increased the container image size to 550MB, exceeding the practical virtualisation capabilities of the 1GB RAM Raspberry Pi 4B models. This led to reimplementing the client scaling using Python threads instead of containers, maintaining functional equivalence while working within hardware constraints.

```
   0x40000df8 <console_init+106>                addi    sp,sp,-16
   0x40000dfa <console_init+108>                li      a2,0
   0x40000dfc <arch_cpu_idle>                   sw      ra,12(sp)
   0x40000dfe <arch_cpu_idle+2>                 jal     0x40000dd6 <console_init+72>
   0x40000e00 <arch_cpu_idle+4>                 .4byte  0x6c2e7777
   0x40000e04 <arch_cpu_idle+8>                 jal     tp,0x400584ea
>  0x40000e08 <arch_cpu_idle+12>                .2byte  0x6f7a
   0x40000e0a <arch_cpu_idle+14>                .2byte  0x6562
   0x40000e0c <arch_cpu_idle+16>                .2byte  0x7474
   0x40000e0e <arch_cpu_idle+18>                lui     t3,0x1a
   0x40000e10 <z_riscv_get_sp_before_exc>       jal     0x400011aa <z_impl_zephyr_fputc+8>
   0x40000e12 <z_riscv_get_sp_before_exc+2>     lui     s0,0xffffa
   0x40000e14 <z_riscv_get_sp_before_exc+4>     .2byte  0x680a
   0x40000e16 <z_riscv_fatal_error_csf>         .2byte  0x7474
   0x40000e18 <z_riscv_fatal_error_csf+2>       .2byte  0x3a70
   0x40000e1a <z_riscv_fatal_error_csf+4>       .4byte  0x77772f2f
   0x40000e1e <z_riscv_fatal_error_csf+8>       .4byte  0x6e672e77
   0x40000e22 <z_riscv_fatal_error_csf+12>      jal     0x400011de <__stdout_hook_install+24>
   0x40000e24 <z_riscv_fatal_error_csf+14>      jal     tp,0x4007811a
   0x40000e28 <z_riscv_fatal_error_csf+18>      csrrsi  t5,0x746,12
   0x40000e2c <z_riscv_fatal_error_csf+22>      .4byte  0x65726177
   0x40000e30 <z_riscv_fatal_error_csf+26>      .4byte  0x6372732f
   0x40000e34 <z_riscv_fatal_error_csf+30>      lui     a6,0xb
   0x40000e36 <z_riscv_fatal_error_csf+32>      lui     a4,0x1a
   0x40000e38 <z_riscv_fatal_error>             .2byte  0x6c68
   0x40000e3a <z_riscv_fatal_error+2>           lui     s0,0xffffa
   0x40000e3c <z_riscv_fatal_error+4>           c.nop   25
   0x40000e3e <z_riscv_fatal_error+6>           unimp
   0x40000e40 <_Fault>                          lw      a5,72(a5)
   0x40000e42 <_Fault+2>                        beqz    a5,0x40000e76 <arch_new_thread+16>
   0x40000e44 <_Fault+4>                        lui     a1,0x40004
   0x40000e48 <_Fault+8>                        li      a2,512
   0x40000e4c <z_irq_spurious+2>                addi    a1,a1,1776
   0x40000e50 <z_irq_spurious+6>                addi    a0,s2,-1744
   0x40000e54 <z_irq_spurious+10>               jal     0x40000d8e <console_init>
   0x40000e56 <z_irq_spurious+12>               lui     a1,0x40005
   0x40000e5a <z_prep_c+2>                      li      a2,64
   0x40000e5e <z_prep_c+6>                      addi    a1,a1,-1808
   0x40000e62 <z_prep_c+10>                     addi    a0,s2,-1744
```

Figure 5.1: GDB Output Showing Spurious Interrupt Errors

5.4.2 Minor Issues

Peripherals Working in LiteX BIOS but not in Buildroot Linux

While I2C and SPI peripherals were functional in the LiteX BIOS, they became inaccessible in Buildroot Linux. This was likely due to device tree mismatches between LiteX and Linux implementations. Given the peripherals weren't critical to the core encryption functionality, resolving this remained a lower priority.

F4PGA Synthesis Toolchain not Completing Synthesis Process

Initial attempts to maintain a fully open-source toolchain using F4PGA (Yosys) for synthesis proved unsuccessful with VexriscvSMP configurations. Rather than debugging the F4PGA compatibility issues, the project pragmatically shifted to Vivado for synthesis, ensuring reliable bitstream generation at the cost of using proprietary tools.

5.5 GitHub Repository

All of the files used in developing this project can be found in my GitHub Repository:

`https://github.com/lcomino64/Thesis`

Although, at the time of submission, the files are unsorted and spread across multiple branches. The branch labelled “pi_vs_pis”, is the most recent branch.

Bibliography

[1] B. A. Forouzan, *TCP/IP Protocol Suite*. McGraw-Hill Education, 5th ed., 2021.

[2] R. Saha, *TELECOMMUNICATIONS ENGINEERING - Technical Note*. M. Eng. (ICT), AIT, THAILAND, 07 2016. https://www.researchgate.net/figure/TCP-and-UDP-Headers_fig56_305506264.

[3] Wikipedia, "Ethernet frame," 2024. https://en.wikipedia.org/wiki/Ethernet_frame.

[4] Garrett Yamasaki, "Three things you should know about ethernet phy," tech. rep., Texas Instruments, 2015.

[5] I. Grigorik, *High Performance Browser Networking*. O'Reilly Media, 2013. Licensed under CC BY-NC-ND 4.0.

[6] Wikipedia, "Advanced encryption standard," Oct 2024. https://en.wikipedia.org/wiki/Advanced_Encryption_Standard.

[7] Enjoy-Digital and LiteX Developers, "LiteX: SoC builder framework." https://github.com/enjoy-digital/litex, 2024.

[8] SFive Inc., *SiFive Interrupt Cookboook V1.2*, February 2020.

[9] Digilent, *Arty A7 Reference Manual*, October 2019. https://digilent.com/reference/programmable-logic/arty-s7/reference-manual.

[10] S. I. Andrew Waterman, Krste Asanovi, *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 2.2*, may 2017.

[11] S. I. Andrew Waterman, Krste Asanovi, *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2*, may 2017.

[12] JAEblog, "Fpga softcore soc shootout," 2020. Just Another Electronics Blog, https://justanotherelectronicsblog.com/?p=705.

[13] T. Alam, "A reliable communication framework and its use in internet of things (iot)," *International Journal of Scientific Research in Computer Science, Engineering and Information Technology*, vol. 3, May 2018.

[14] M. Michael, "Attack landscape h1 2019: Iot, smb traffic abound," September 2019.

[15] M. Frustaci, P. Pace, G. Aloi, and G. Fortino, "Evaluating critical security issues of the iot world: Present and future challenges," *IEEE Internet of Things Journal*, vol. 5, pp. 2483–2495, October 2017.

[16] A. Shilov, "Nvidia to ship a billion of RISC-V cores in 2024," 2024.

[17] J. F. Kurose and K. W. Ross, *Computer Networking: A Top-Down Approach*. Pearson, 8th ed., 2021.

[18] Xilinx, *Tri-Mode Ethernet MAC LogiCORE IP Product Guide*, 2023. PG051.

[19] Xilinx, *AXI 1G/2.5G Ethernet Subsystem Product Guide*, 2024. PG138.

[20] Xilinx, *AXI Ethernet Lite MAC LogiCORE IP Product Guide*, 2021. PG135.

[21] Wikipedia, "Media-independent interface," Mar 2024. `https://en.wikipedia.org/wiki/Media-independent_interface`.

[22] Wikipedia, "Small form-factor pluggable," Mar 2024. `https://en.wikipedia.org/wiki/Small_Form-factor_Pluggable`.

[23] T. Isobe, S. Tsutsumi, K. Seto, K. Aoshima, and K. Kariya, "10gbps implementation of TLS/SSL accelerator on FPGA," in *2010 International Conference on Quality, Reliability, Risk, Maintenance, and Safety Engineering*, pp. 1–6, IEEE, 2010.

[24] G. Restuccia, H. Tschofenig, and E. Baccelli, "Low-power IoT communication security: On the performance of DTLS and TLS 1.3," *IEEE Access*, 2020.

[25] N. G. Tsoutsos and M. Maniatakos, "Anatomy of memory corruption attacks and mitigations in embedded systems," *IEEE Embedded Systems Letters*, vol. 10, no. 3, pp. 95–98, 2018.

[26] Z. Shao, Q. Zhuge, Y. He, and E.-M. Sha, "Defending embedded systems against buffer overflow via hardware/software," in *19th Annual Computer Security Applications Conference, 2003. Proceedings.*, pp. 352–361, 2003.

[27] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3." RFC 8446, Aug. 2018. `https://www.rfc-editor.org/info/rfc8446`.

[28] National Institute of Standards and Technology, "Advanced encryption standard (aes)," Federal Information Processing Standards Publication FIPS 197, National Institute of Standards and Technology, Gaithersburg, MD, May 2023. Updated May 9, 2023.

[29] The OpenSSL Project, *OpenSSL Documentation*, 2023. Open source cryptography and SSL/TLS toolkit.

[30] D. Patterson and A. Waterman, "RISC-V: An open standard for SoCs," 7 2014.

[31] A. Waterman and D. Patterson, *The RISC-V Reader: An Open Architecture Atlas*. Strawberry Canyon, 2017.

[32] K. Asanović and D. A. Patterson, "Instruction sets should be free: The case for RISC-V," Technical Report UCB/EECS-2014-146, EECS Department, University of California, Berkeley, 2014.

[33] T. Lu, "A survey on risc-v security: Hardware and architecture," 07 2021.

[34] C. Maxfield, "Fundamentals of FPGAs - Part 4: Getting started with Xilinx FPGAs," *DigiKey Electronics*, April 2020. Contributed By DigiKey's North American Editors.

[35] R.-V. Forum and Help, "Risc-v ip cores," 2024. `https://www.riscfive.com/risc-v-ip-cores/`.

[36] P. Anemaet and T. van As, "Microprocessor soft-cores: An evaluation of design methods and concepts on FPGAs," Tech. Rep. ET4 078, Delft University of Technology, 2008. Computer Architecture Special Topics.

[37] C. Papon, F. Kermarrec, and Contributors, "VexRiscv: A FPGA friendly 32 bit RISC-V CPU implementation." `https://github.com/SpinalHDL/VexRiscv`, 2024.

[38] C. Papon and Contributors, "SpinalHDL: A high-level hardware description language." `https://github.com/SpinalHDL/SpinalHDL`, 2024.

[39] G. Marini and Contributors, "openfpgaloader - universal FPGA loader," 2024. Universal and cross-platform FPGA loader supporting numerous FPGA families.

[40] Enjoy-Digital and Contributors, "Zephyr on LiteX VexRiscv," 2024. Open source repository for running Zephyr RTOS on LiteX VexRiscv.

[41] Enjoy-Digital and Contributors, "Linux on LiteX VexRiscv," 2024. Open source repository for running Linux on LiteX VexRiscv.

[42] B. Developers, "Buildroot - making embedded linux easy," 2024. Open source build system for embedded Linux systems, `https://buildroot.org/`.

[43] E. Andersen and B. Developers, "Busybox: The swiss army knife of embedded linux," 2024. Combines tiny versions of many common UNIX utilities into a single small executable, `https://www.busybox.net/`.

[44] W. Digital and Contributors, "Opensbi - open source supervisor binary interface," 2024. Reference implementation of the RISC-V SBI specification, `https://github.com/riscv/opensbi`.

[45] Raspberry Pi Foundation, "Raspberry pi 4 tech specs," 2024.

Appendix A

Appendix

A.1 Full Single Core Results

Figure A.1: Single Core, Basic Test 1 Server Metrics

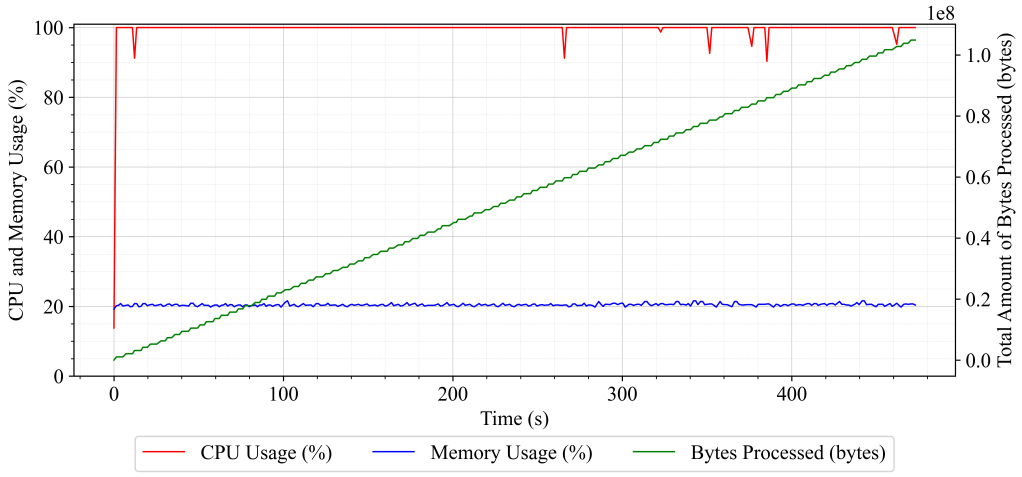
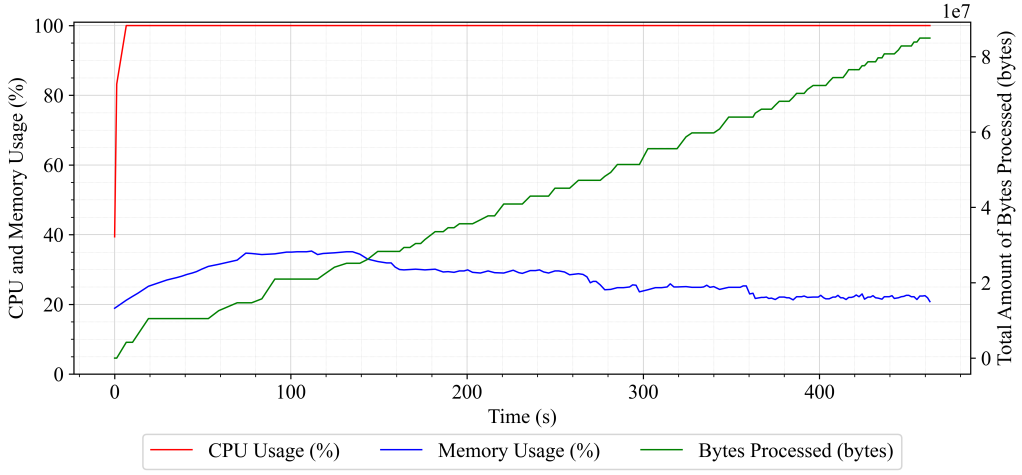


Figure A.2: Single Core, Basic Test 2 Server and Client Metrics



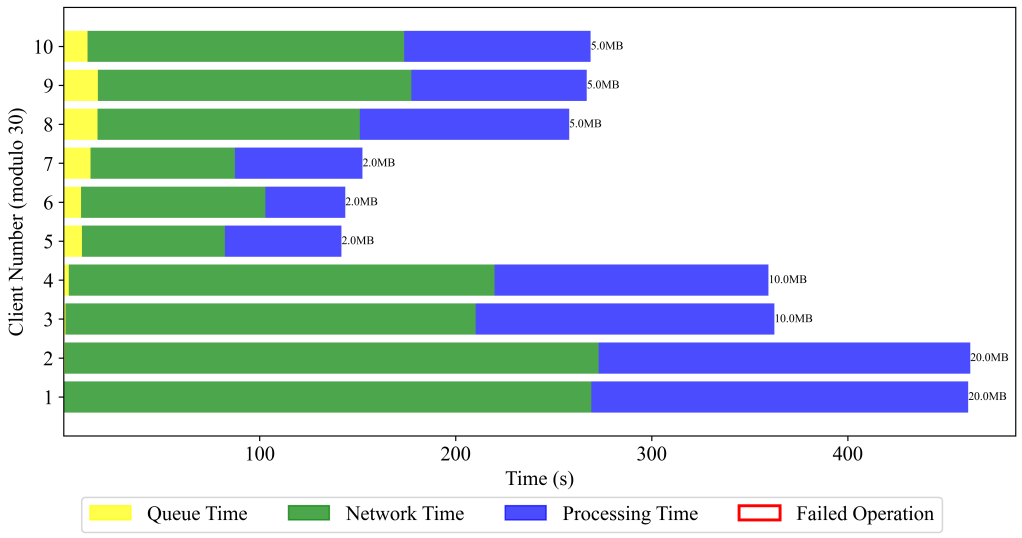


Figure A.3: Single Core, Basic Test 3 Server and Client Metrics

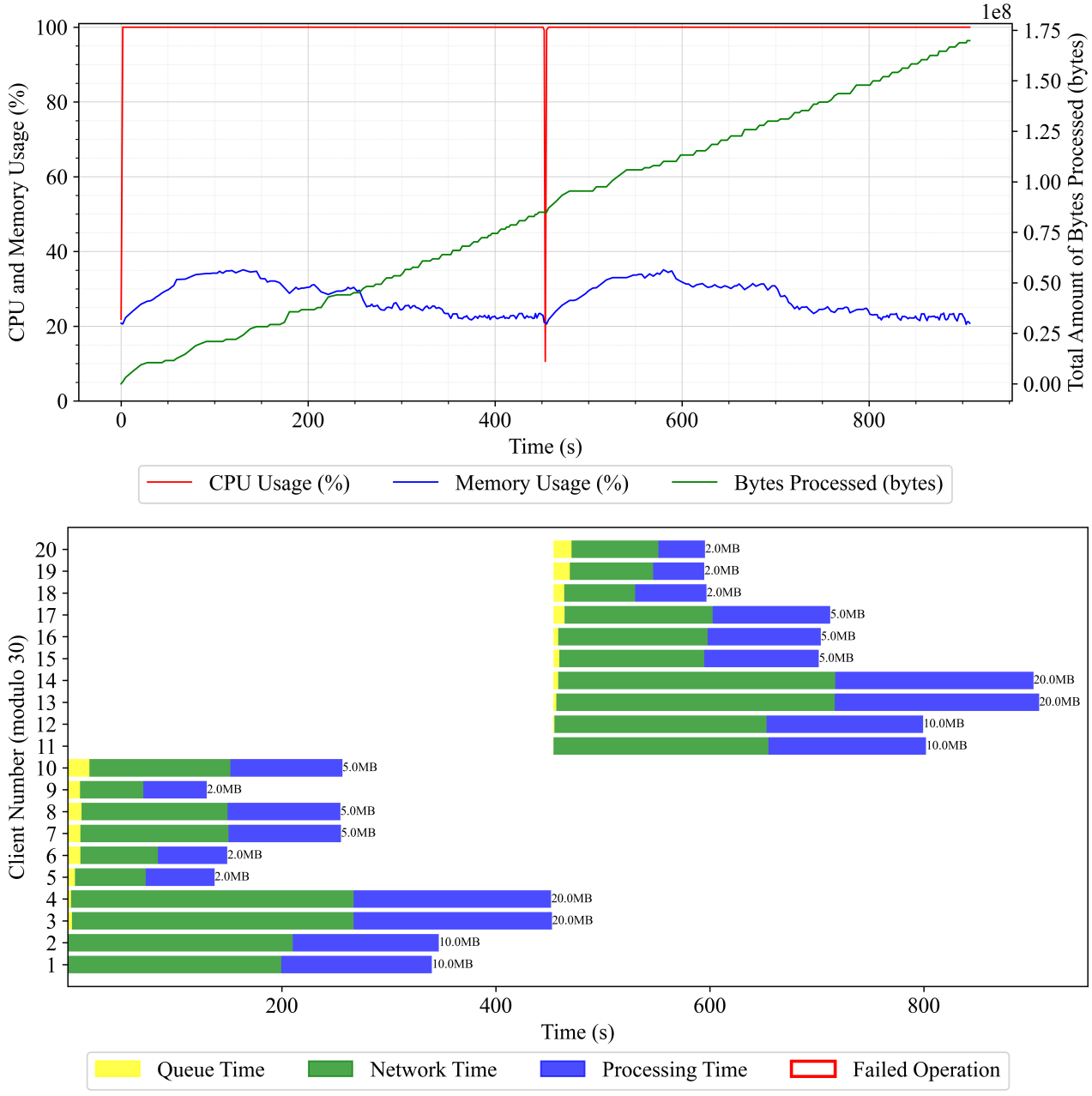
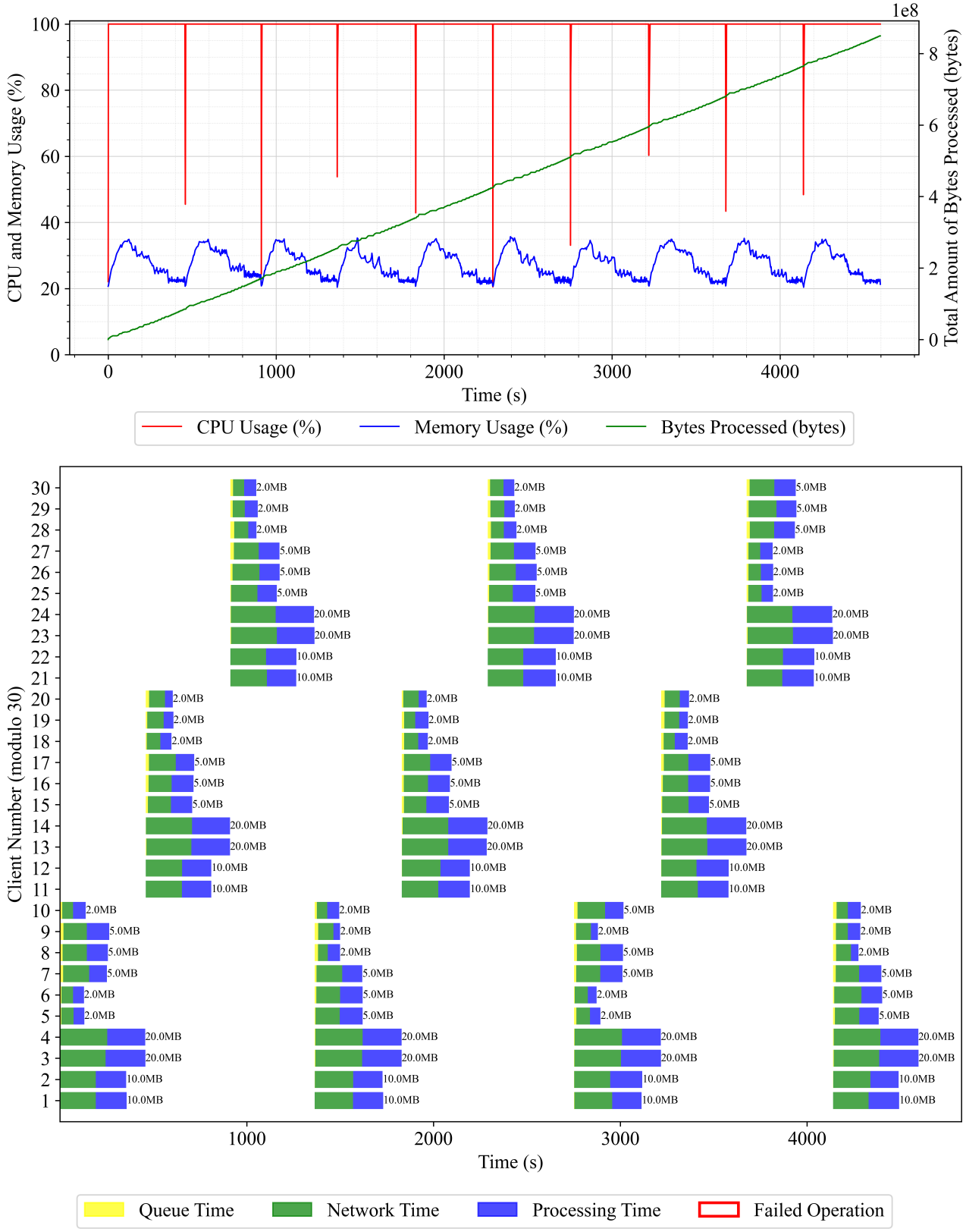


Figure A.4: Single Core, Large Scale, 100 clients, Server and Client Metrics



A.2 Full Dual Core Results

Figure A.5: Dual Core, Basic Test 1 Server Metrics

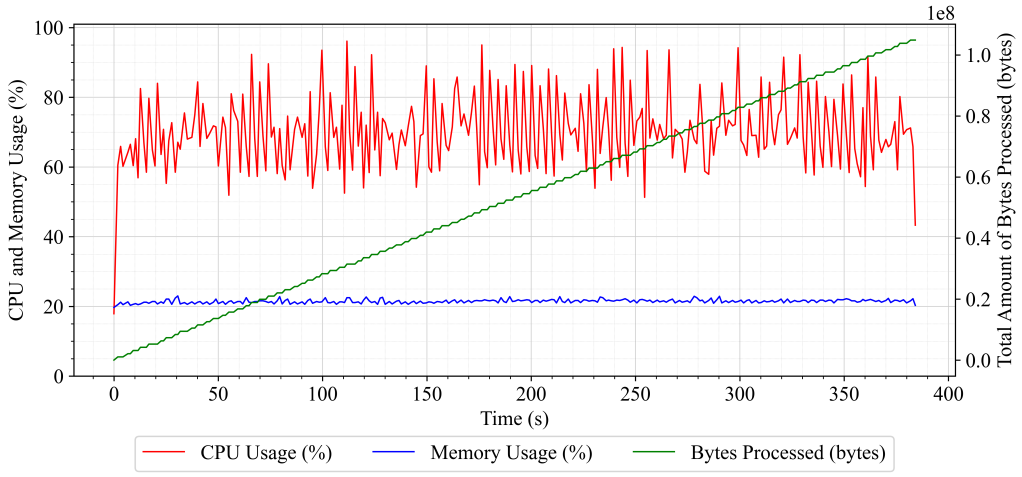


Figure A.6: Dual Core, Basic Test 2 Server and Client Metrics

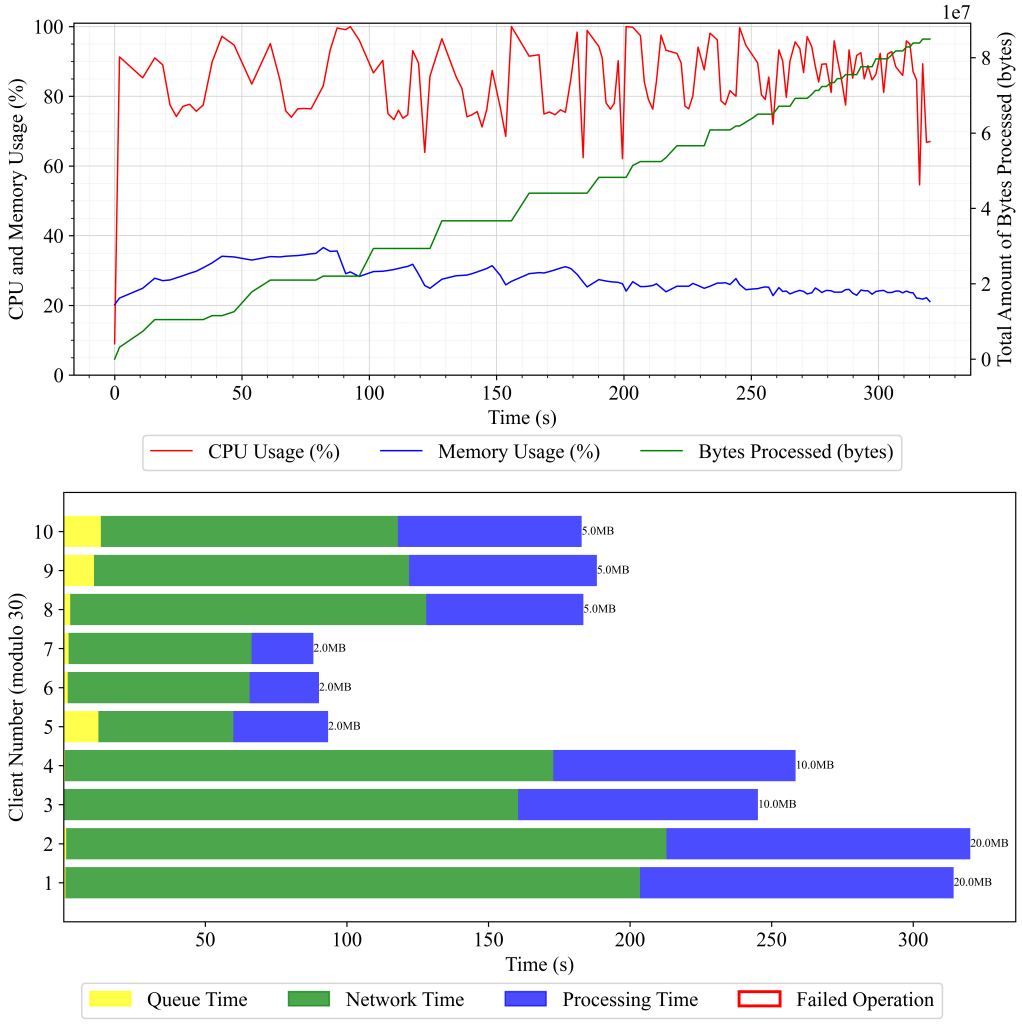


Figure A.7: Dual Core, Basic Test 3 Server and Client Metrics

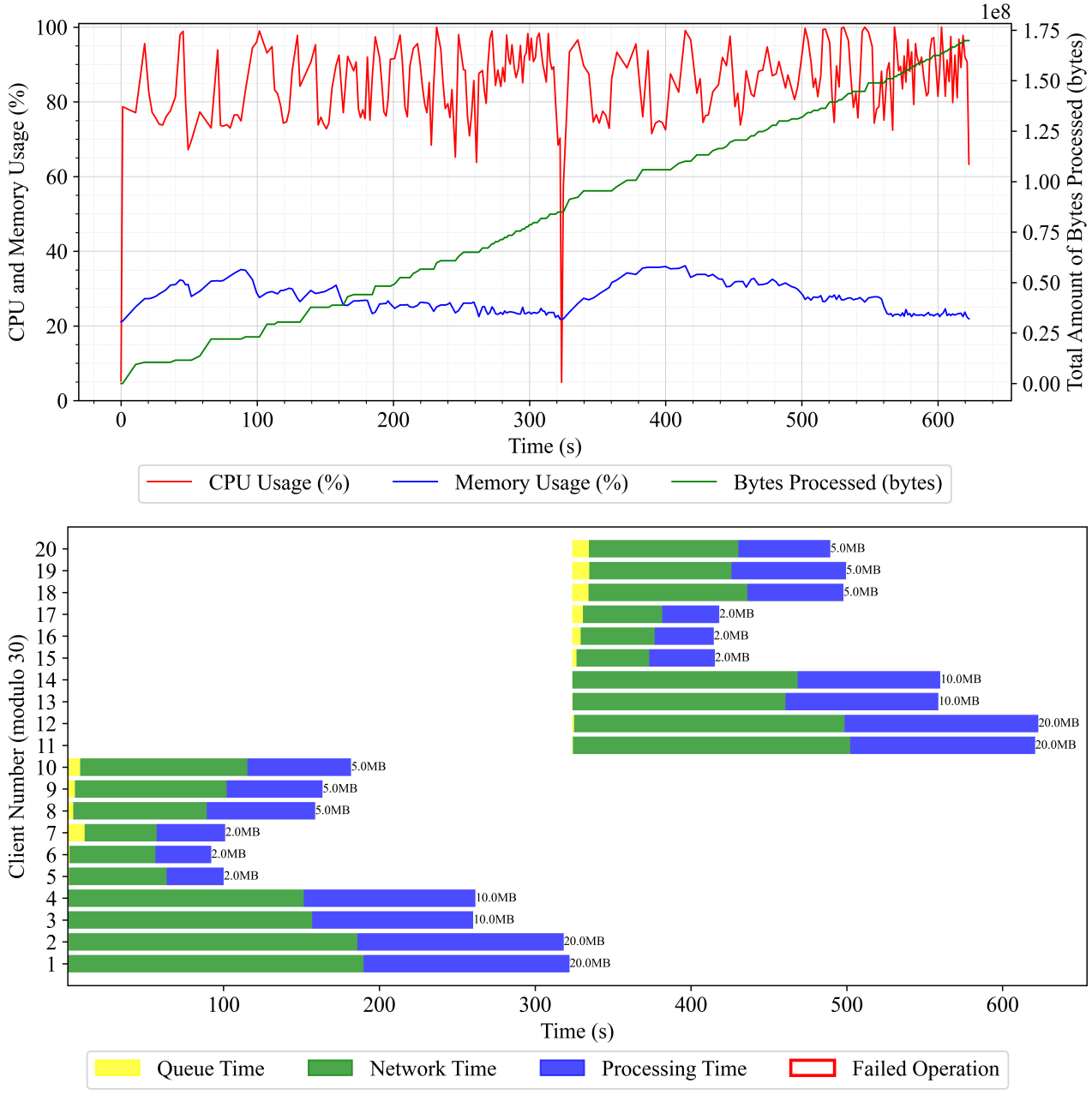
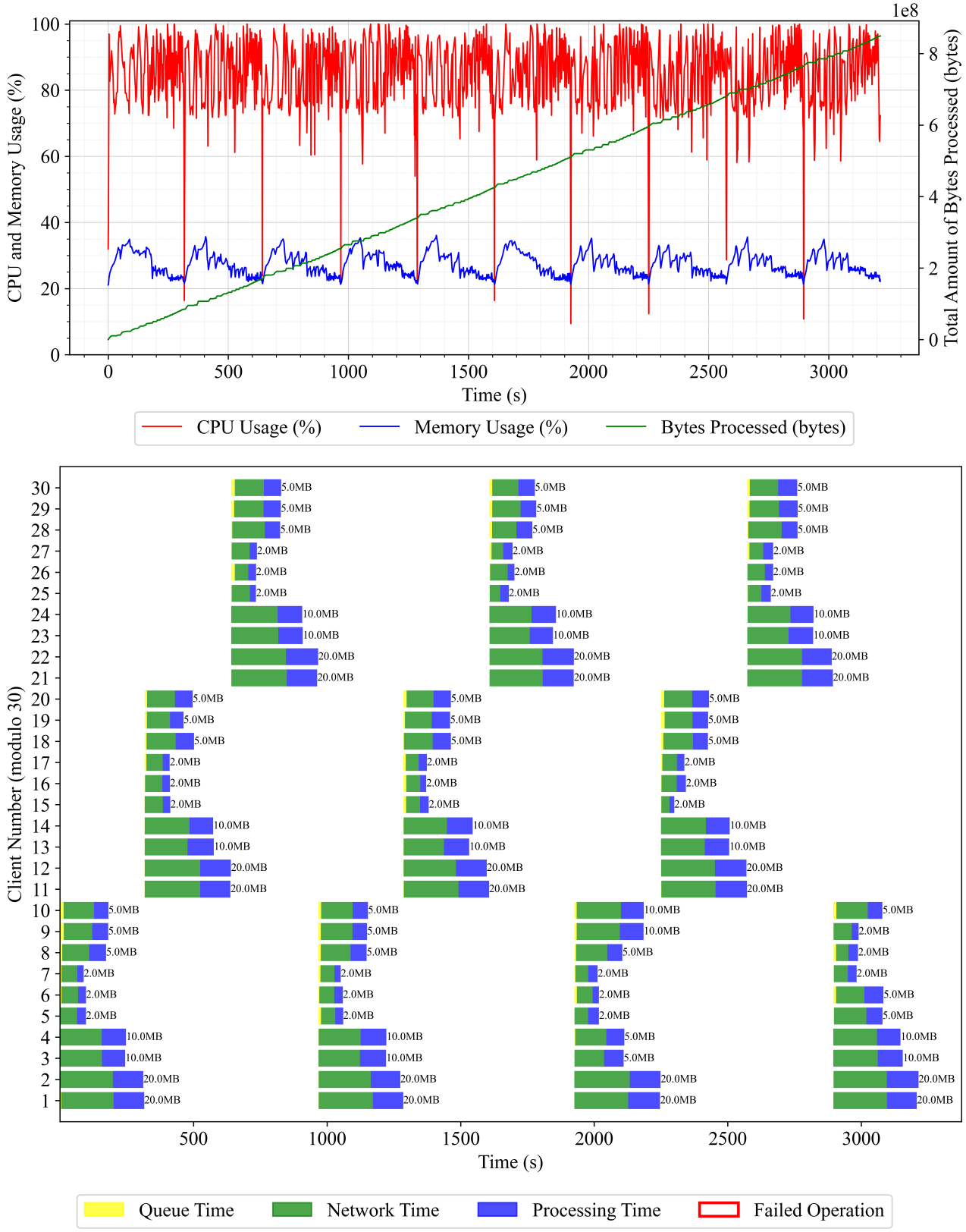


Figure A.8: Dual Core, Large Scale, 100 clients, Server and Client Metrics



A.3 Full Quad Core Results

Figure A.9: Quad Core, Basic Test 1 Server Metrics

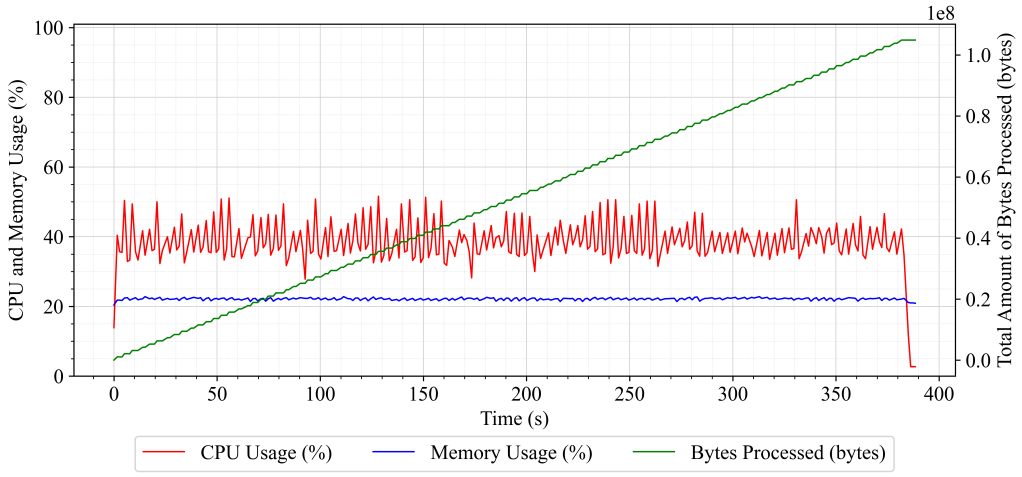
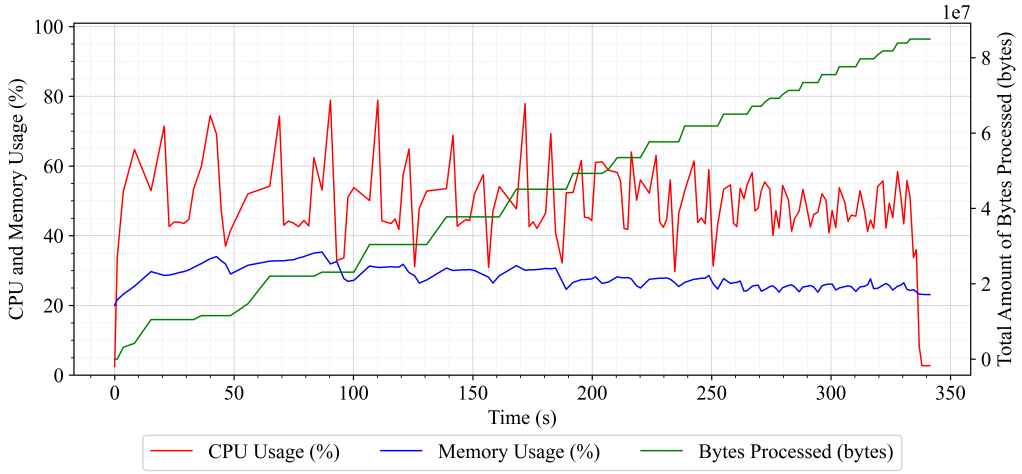


Figure A.10: Quad Core, Basic Test 2 Server and Client Metrics



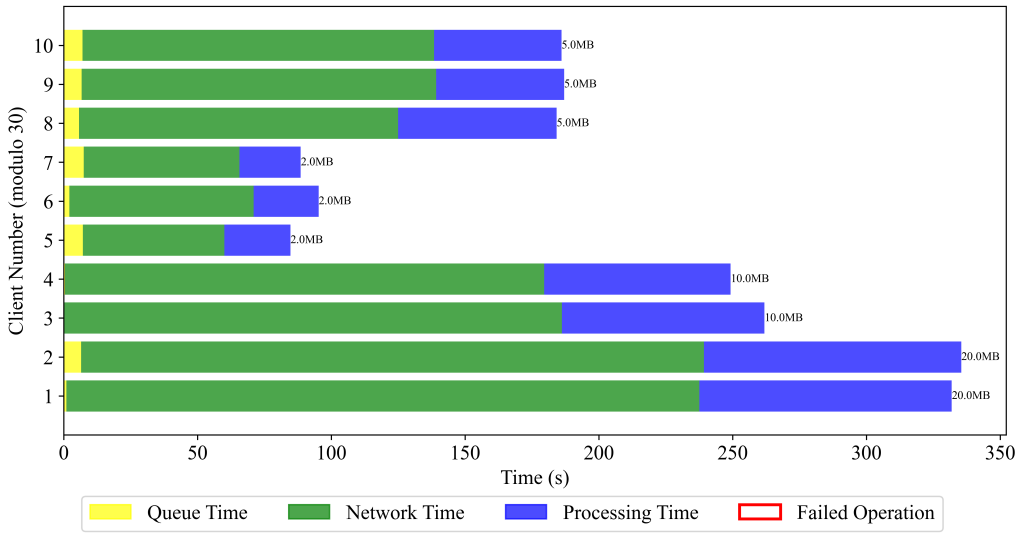


Figure A.11: Quad Core, Basic Test 3 Server and Client Metrics

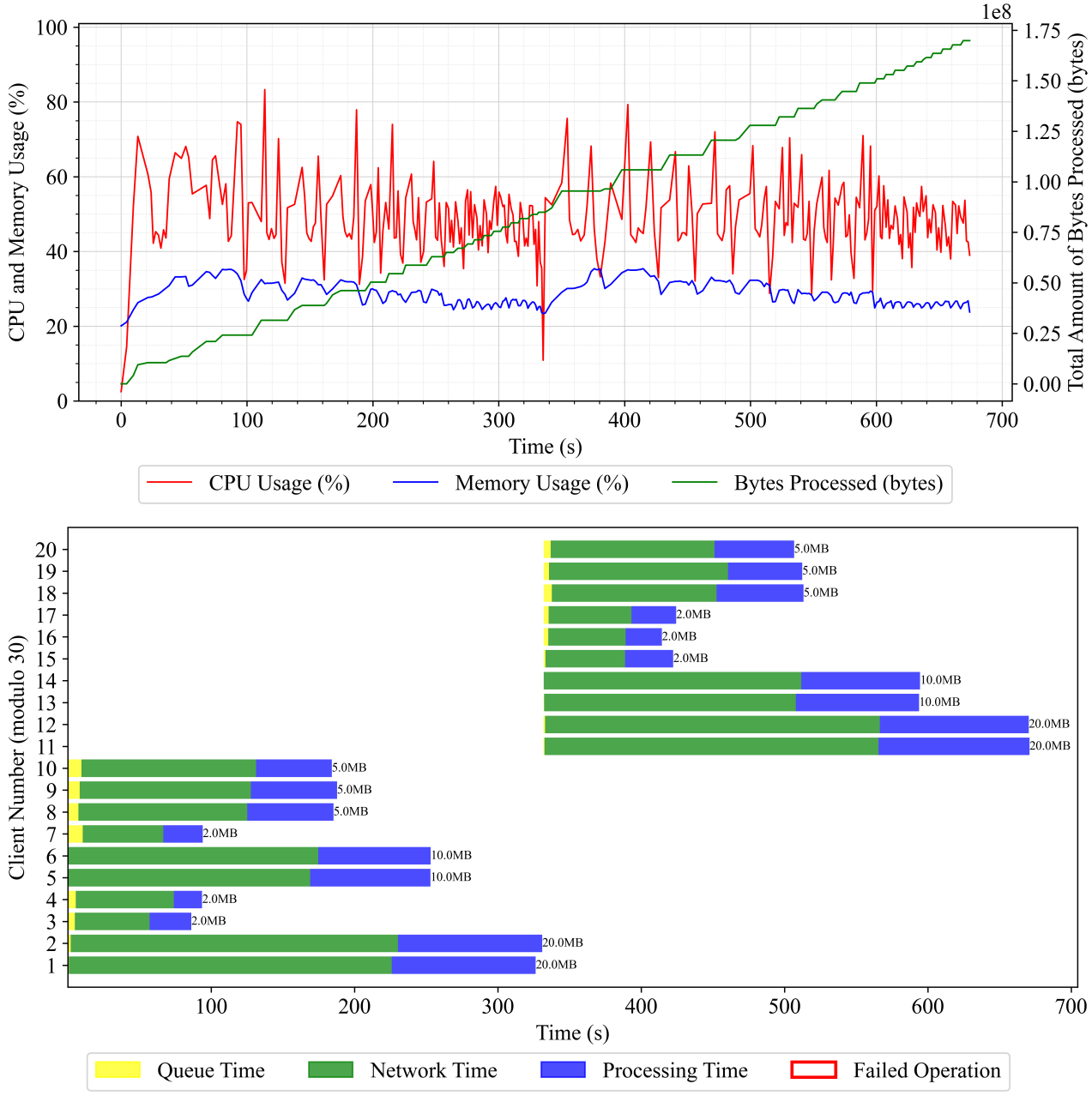
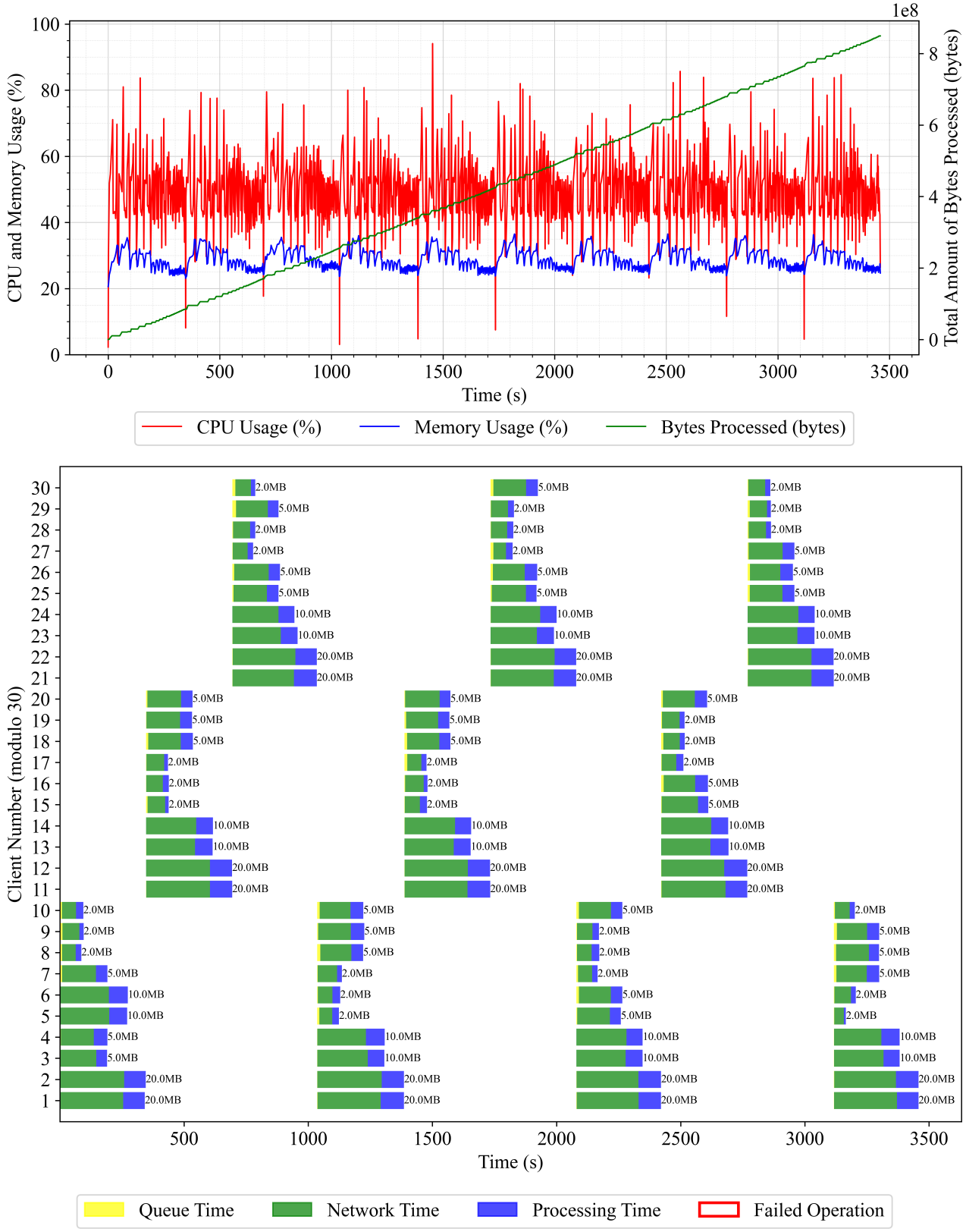


Figure A.12: Quad Core, Large Scale, 100 clients, Server and Client Metrics



A.4 Full Raspberry Pi Results

Figure A.13: Raspberry Pi, Basic Test 1 Server Metrics

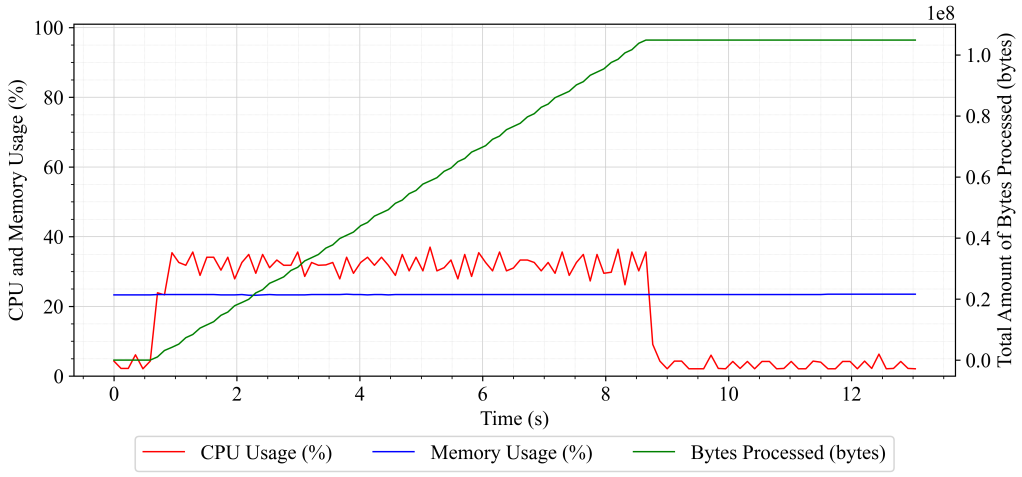
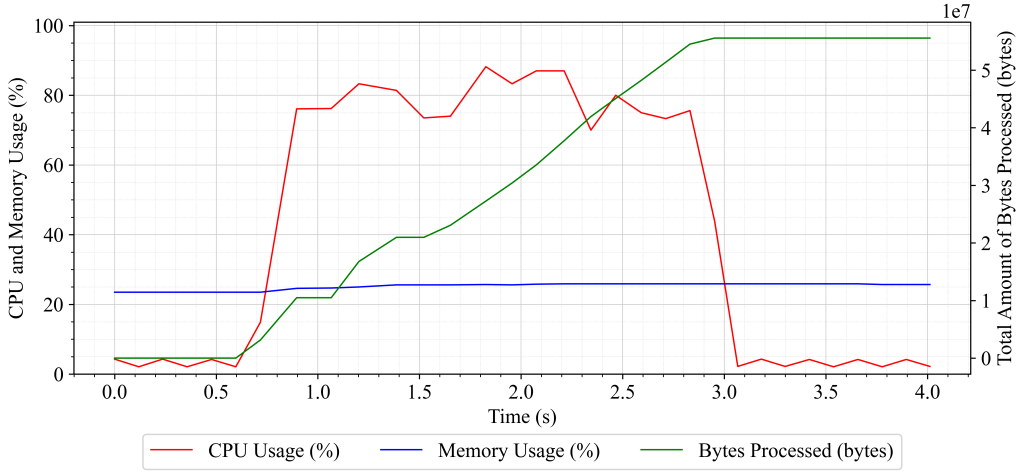


Figure A.14: Raspberry Pi, Basic Test 2 Server and Client Metrics



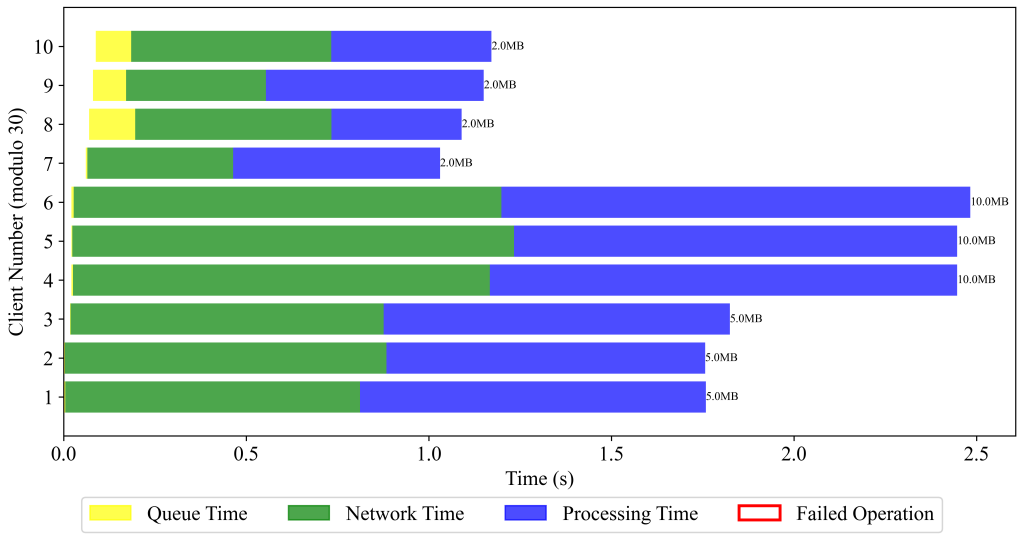


Figure A.15: Raspberry Pi, Basic Test 3 Server and Client Metrics

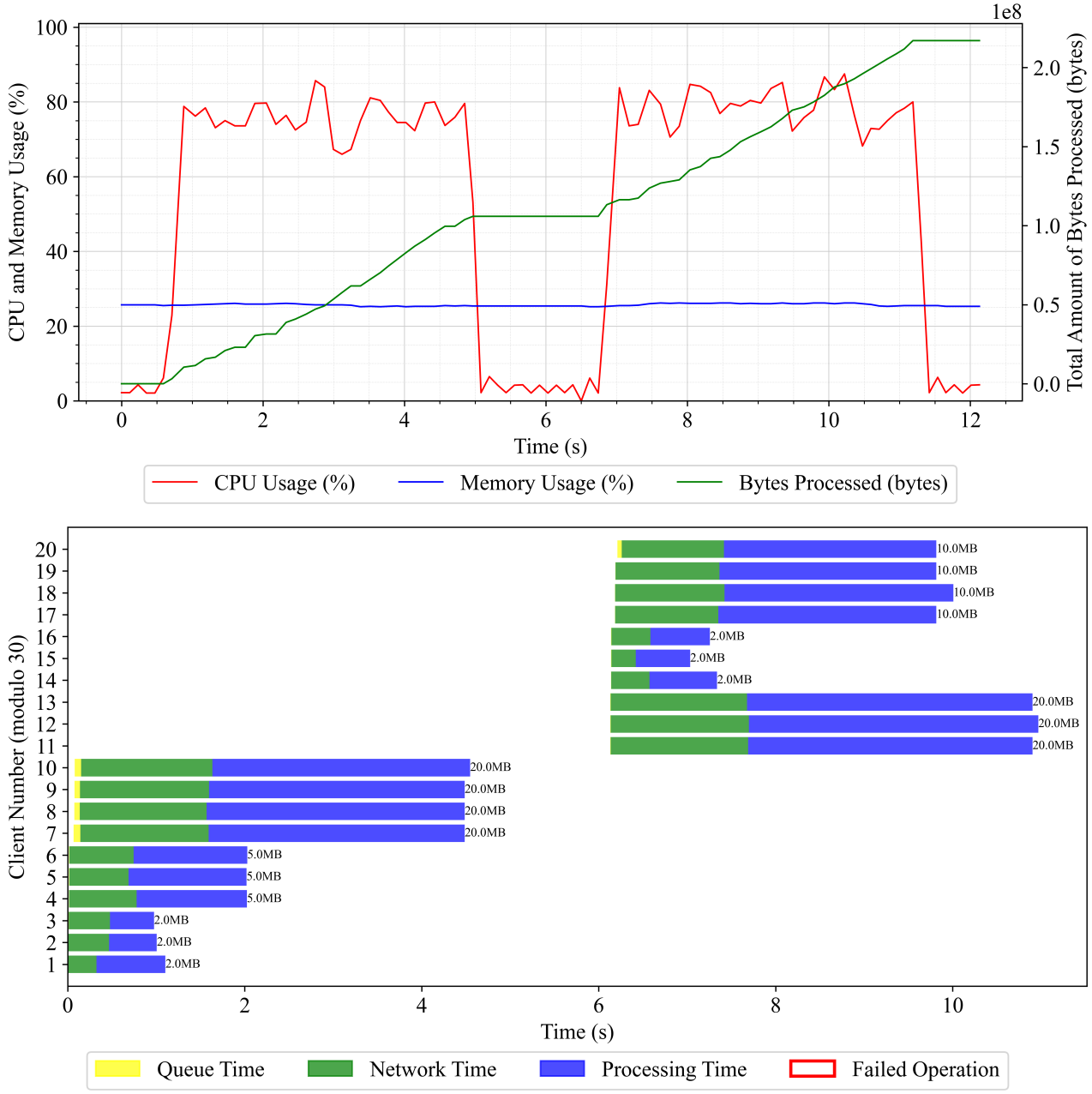
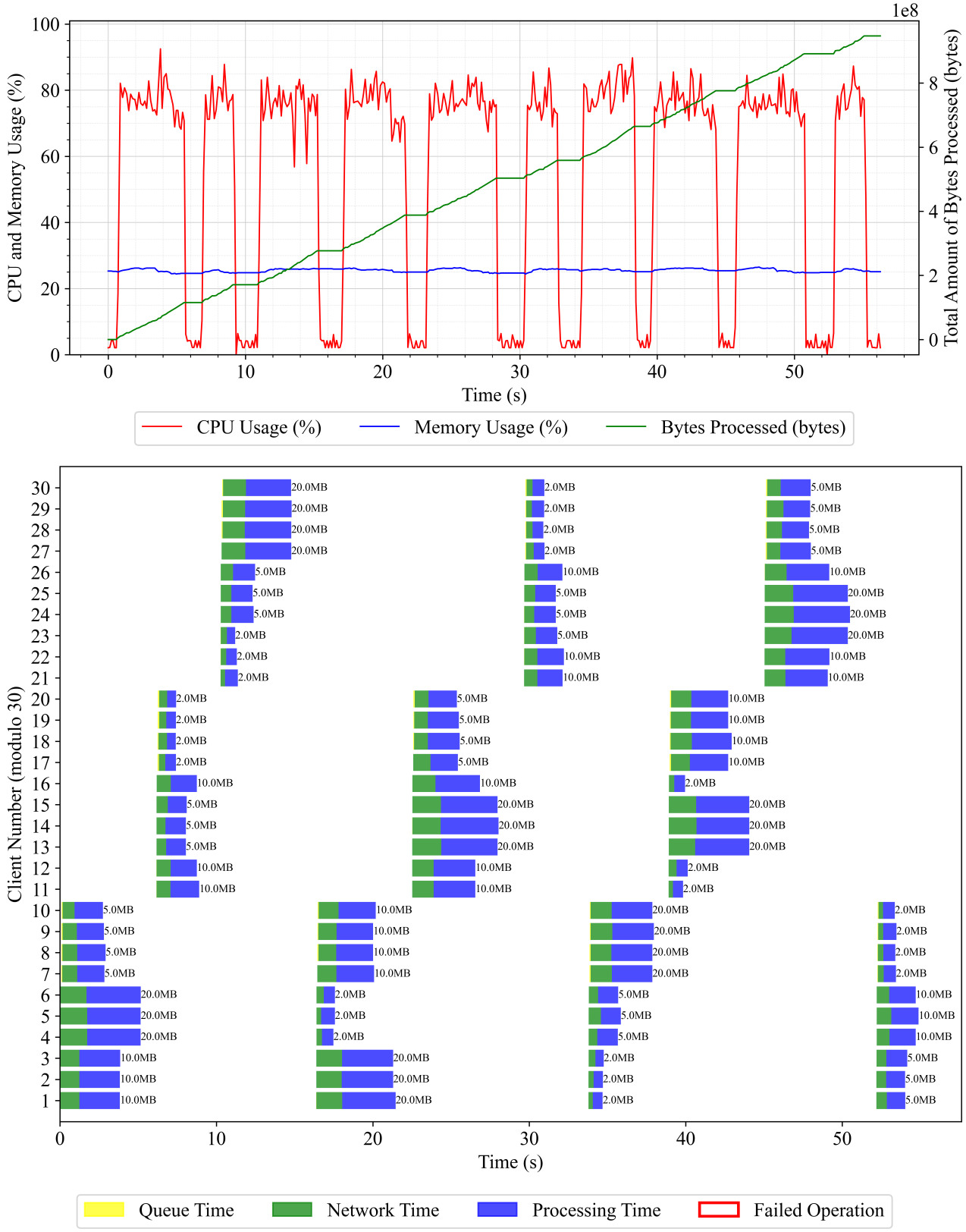


Figure A.16: Raspberry Pi, Large Scale, 100 clients, Server and Client Metrics



A.5 Full Improved Dual Core Results

Figure A.17: Improved Dual Core, Basic Test 1 Server Metrics

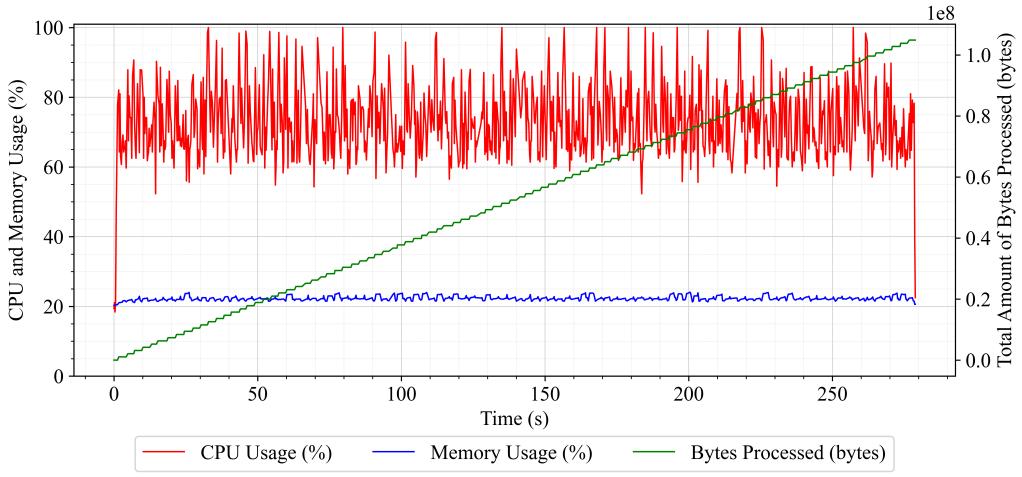
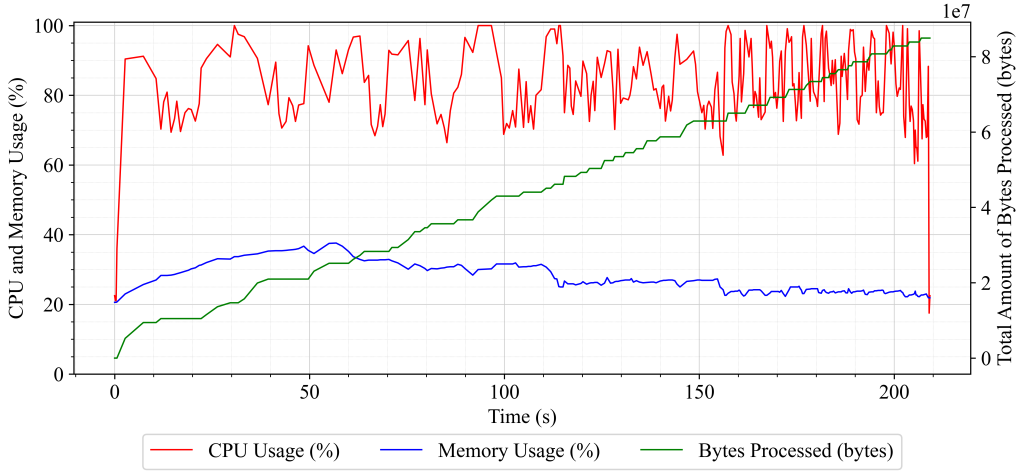


Figure A.18: Improved Dual Core, Basic Test 2 Server and Client Metrics



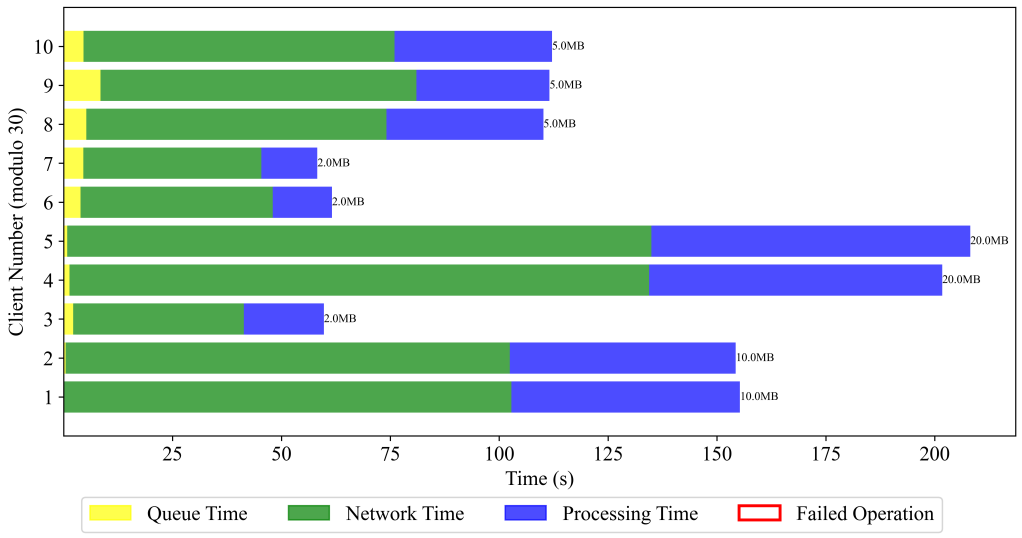


Figure A.19: Improved Dual Core, Basic Test 3 Server and Client Metrics

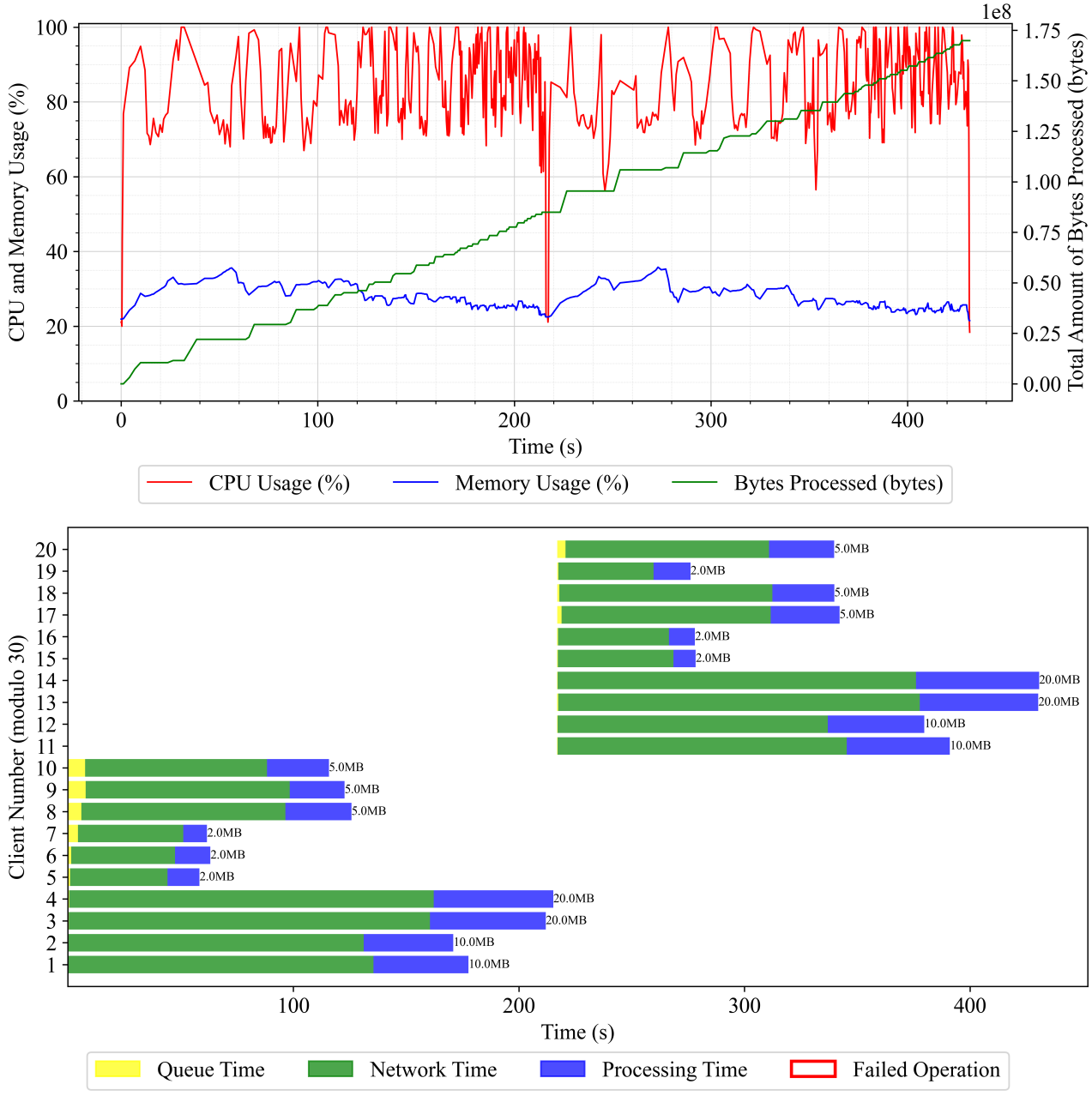
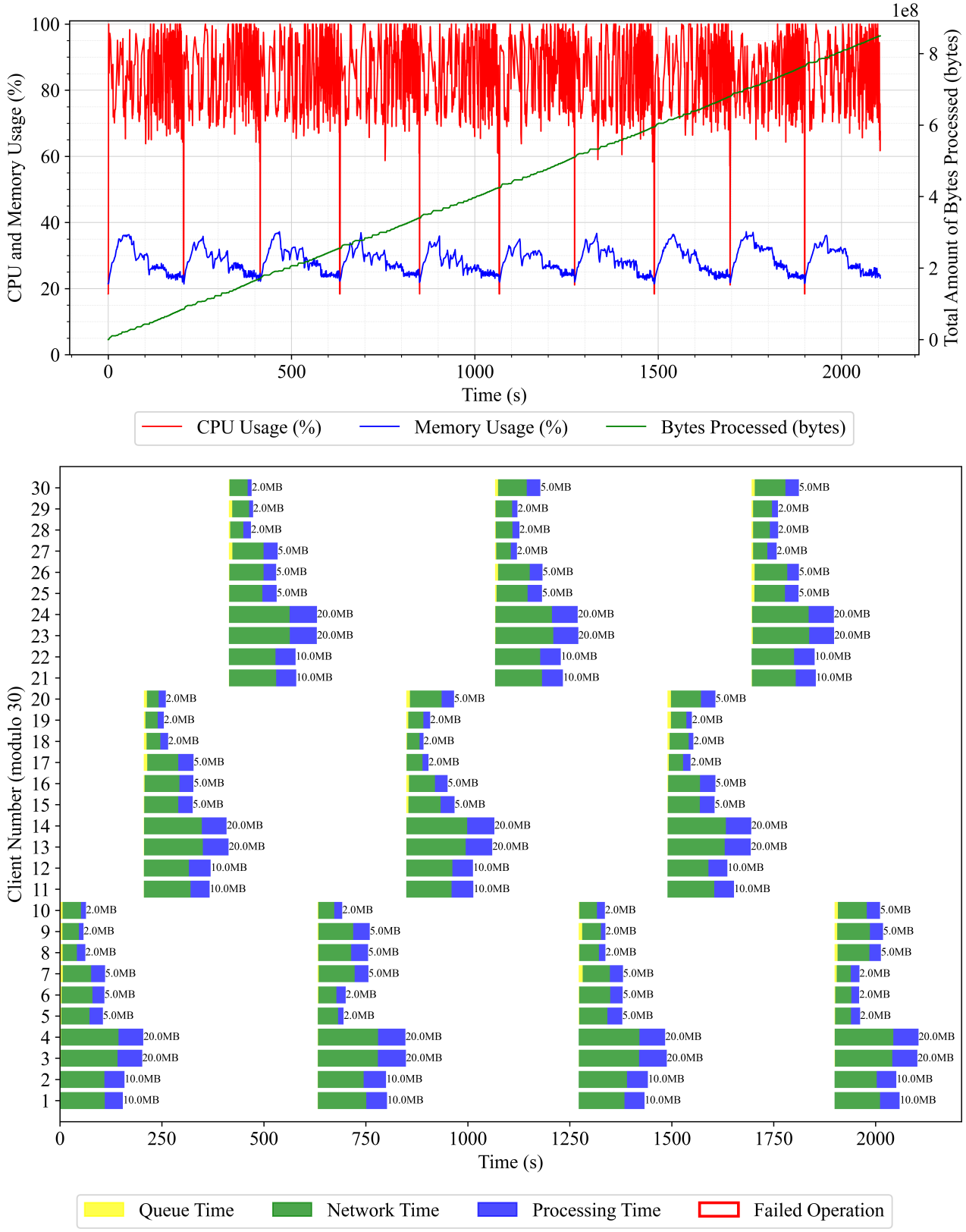


Figure A.20: Improved Dual Core, Large Scale, 100 clients, Server and Client Metrics



A.6 LiteX BIOS Console Help Menu

```
--============== Console ================--

litex> help

LiteX BIOS, available commands:

leds - Set Leds value
flush_l2_cache - Flush L2 cache
flush_cpu_dcache - Flush CPU data cache
crc - Compute CRC32 of a part of the address space
uptime - Uptime of the system since power-up
ident - Identifier of the system
help - Print this help

netboot - Boot via Ethernet (TFTP)
serialboot - Boot from Serial (SFL)
reboot - Reboot
boot - Boot from Memory

mem_cmp - Compare memory content
mem_speed - Test memory speed
mem_test - Test memory access
mem_copy - Copy address space
mem_write - Write address space
mem_read - Read address space
mem_list - List available memory regions

flash_erase_range - Erase flash range
flash_from_sdcard - Write file from SD card to flash
flash_write - Write to flash

i2c_dev - List/Set I2C controller(s)
i2c_scan - Scan for I2C slaves
i2c_read - Read over I2C
i2c_write - Write over I2C
i2c_reset - Reset I2C line state

sdram_spd - Read SDRAM SPD EEPROM
sdram_mr_write - Write SDRAM Mode Register
sdram_cal - Calibrate SDRAM
```

```
sdram_test - Test SDRAM
sdram_init - Initialize SDRAM (Init + Calibration)
sdram_force_wrphase - Force write phase
sdram_force_rdphase - Force read phase

mdio_dump - Dump MDIO registers
mdio_read - Read MDIO register
mdio_write - Write MDIO register
```

A.7 Full Output From Loading a Zephyr Binary

```
lachlancomino@ubuntu-machine:~/zephyrproject/zephyr$ litex_term /dev/ttyUSB2
 --speed 115200 --kernel build/zephyr/zephyr.bin
    / / (_) /____ | |/_/
     / /__/ / __/ -_)> <
    /____/_/\__/\__/_/|_|
  Build your hardware, easily!

 (c) Copyright 2012-2024 Enjoy-Digital
 (c) Copyright 2007-2015 M-Labs

 BIOS CRC passed (02a8afa1)

 LiteX git sha1: ef775e0b8

--================ SoC ==================--
CPU:  VexRiscv SMP-STANDARD @ 100MHz
BUS:  wishbone 32-bit @ 4GiB
CSR:  32-bit data
ROM:  64.0KiB
SRAM:  16.0KiB
L2:  8.0KiB
SDRAM:  64.0MiB 16-bit @ 800MT/s (CL-7 CWL-5)
MAIN-RAM: 256.0MiB

--=========== Initialization ============--
Initializing SDRAM @0x40000000...
Switching SDRAM to software control.
Read leveling:
  m0, b00: |00000000000000000000000000000000| delays: -
  m0, b01: |00000000000000000000000000000000| delays: -
  m0, b02: |11111111111000000000000000000000| delays: 05+-05
  m0, b03: |00000000000011111111111111100000| delays: 19+-06
  m0, b04: |00000000000000000000000000000011| delays: 30+-00
  m0, b05: |00000000000000000000000000000000| delays: -
  m0, b06: |00000000000000000000000000000000| delays: -
  m0, b07: |00000000000000000000000000000000| delays: -
  best: m0, b03 delays: 19+-06
  m1, b00: |00000000000000000000000000000000| delays: -
  m1, b01: |00000000000000000000000000000000| delays: -
  m1, b02: |11111111111000000000000000000000| delays: 05+-05
```

```
  m1, b03: |00000000000001111111111111100000| delays: 20+-06
  m1, b04: |00000000000000000000000000000011| delays: 30+-00
  m1, b05: |00000000000000000000000000000000| delays: -
  m1, b06: |00000000000000000000000000000000| delays: -
  m1, b07: |00000000000000000000000000000000| delays: -
  best: m1, b03 delays: 19+-06
Switching SDRAM to hardware control.
Memtest at 0x40000000 (2.0MiB)...
  Write: 0x40000000-0x40200000 2.0MiB
   Read: 0x40000000-0x40200000 2.0MiB
Memtest OK
Memspeed at 0x40000000 (Sequential, 2.0MiB)...
  Write speed: 161.0MiB/s
   Read speed: 77.9MiB/s

--============== Boot ==================--
Booting from serial...
Press Q or ESC to abort boot completely.
sL5DdSMmkekro
[LITEX-TERM] Received firmware download request from the device.
[LITEX-TERM] Uploading build/zephyr/zephyr.bin to 0x40000000 (33982 bytes)...
[LITEX-TERM] Upload calibration... (inter-frame: 10.00us, length: 64)
[LITEX-TERM] Upload complete (9.9KB/s).
[LITEX-TERM] Booting the device.
[LITEX-TERM] Done.
Executing booted program at 0x40000000

--============== Liftoff! ===============--
```

A.8 Full Output from Loading Buildroot Linux

```
Terminal ready
    Build your hardware, easily!

    (c) Copyright 2012-2024 Enjoy-Digital
    (c) Copyright 2007-2015 M-Labs

    BIOS CRC passed (1df78940)

    LiteX git sha1: a350d2e90

--=============== SoC ==================--
CPU:  VexRiscv SMP-LINUX @ 150MHz
BUS:  wishbone 32-bit @ 4GiB
CSR:  32-bit data
ROM:  64.0KiB
SRAM:  16.0KiB
L2:  8.0KiB
FLASH:  16.0MiB
SDRAM:  64.0MiB 16-bit @ 1200MT/s (CL-11 CWL-7)
MAIN-RAM: 256.0MiB

--=========== Initialization ============--
Ethernet init...

Initializing SDRAM @0x40000000...
Switching SDRAM to software control.
Read leveling:
    m0, b00: |00000000000000000000000000000000| delays: -
    m0, b01: |00000000000000000000000000000000| delays: -
    m0, b02: |00000000000000000000000000000000| delays: -
    m0, b03: |11111110000000000000000000000000| delays: 02+-02
    m0, b04: |00000000111111110000000000000000| delays: 12+-04
    m0, b05: |00000000000000000001111111100000| delays: 22+-03
    m0, b06: |00000000000000000000000000000001| delays: -
    m0, b07: |00000000000000000000000000000000| delays: -
    best: m0, b04 delays: 12+-04
    m1, b00: |00000000000000000000000000000000| delays: -
    m1, b01: |00000000000000000000000000000000| delays: -
    m1, b02: |00000000000000000000000000000000| delays: -
    m1, b03: |11111100000000000000000000000000| delays: 02+-02
```

```
    m1, b04: |00000000011111111000000000000000| delays: 12+-04
    m1, b05: |00000000000000000001111111110000| delays: 23+-03
    m1, b06: |00000000000000000000000000000001| delays: 30+-00
    m1, b07: |00000000000000000000000000000000| delays: -
    best: m1, b05 delays: 23+-04
Switching SDRAM to hardware control.
Memtest at 0x40000000 (2.0MiB)...
    Write: 0x40000000-0x40200000 2.0MiB
    Read: 0x40000000-0x40200000 2.0MiB
Memtest OK
Memspeed at 0x40000000 (Sequential, 2.0MiB)...
    Write speed: 242.4MiB/s
    Read speed: 116.8MiB/s

Initializing s25fl128l SPI Flash @0x01000000...
Enabling Quad mode...
SPI Flash clk configured to 18 MHz
Memspeed at 0x1000000 (Sequential, 4.0KiB)...
    Read speed: 30.9MiB/s
Memspeed at 0x1000000 (Random, 4.0KiB)...
    Read speed: 5.8MiB/s

--============== Boot ==================--
Booting from serial...
Press Q or ESC to abort boot completely.
sL5DdSMmkekro
Timeout
Booting from network...
Local IP: 192.168.1.50
Remote IP: 192.168.1.100
Booting from boot.json...
Copying Image to 0x40000000... (8510016 bytes)
Copying rv32.dtb to 0x40ef0000... (5694 bytes)
Copying rootfs.cpio to 0x41000000... (33890816 bytes)
Copying opensbi.bin to 0x40f00000... (263652 bytes)
Executing booted program at 0x40f00000

--============== Liftoff! ===============--

OpenSBI v1.3
    ---- ----- ---- -----
```

```
   / __ \ / ____| _ \_ _|
   | | | |_ __ ___ _ __ | (___ | |_) || |
   | | | | '_ \ / _ \ '_ \ \___ \| _ < | |
   | |__| | |_) | __/ | | |____) | |_) || |_
   \____/| .__/ \___|_| |_|_____/|____/_____|
       | |
       |_|
Platform Name : LiteX / VexRiscv-SMP
Platform Features : medeleg
Platform HART Count : 8
Platform IPI Device : aclint-mswi
Platform Timer Device : aclint-mtimer @ 100000000Hz
Platform Console Device : litex_uart
Platform HSM Device : ---
Platform PMU Device : ---
Platform Reboot Device : ---
Platform Shutdown Device : ---
Platform Suspend Device : ---
Platform CPPC Device : ---
Firmware Base : 0x40f00000
Firmware Size : 376 KB
Firmware RW Offset : 0x40000
Firmware RW Size : 120 KB
Firmware Heap Offset : 0x52000
Firmware Heap Size : 48 KB (total), 3 KB (reserved), 8 KB (used), 36 KB (free)
Firmware Scratch Size : 4096 B (total), 452 B (used), 3644 B (free)
Runtime SBI Version : 1.0

Domain0 Name : root
Domain0 Boot HART : 0
Domain0 HARTs : 0*,1*,2*,3*,4*,5*,6*,7*
Domain0 Region00 : 0xf0018000-0xf001bfff M: (I,R,W) S/U: ()
Domain0 Region01 : 0xf0010000-0xf0017fff M: (I,R,W) S/U: ()
Domain0 Region02 : 0x40f40000-0x40f5ffff M: (R,W) S/U: ()
Domain0 Region03 : 0x40f00000-0x40f3ffff M: (R,X) S/U: ()
Domain0 Region04 : 0x00000000-0xffffffff M: (R,W,X) S/U: (R,W,X)
Domain0 Next Address : 0x40000000
Domain0 Next Arg1 : 0x40ef0000
Domain0 Next Mode : S-mode
Domain0 SysReset : yes
Domain0 SysSuspend : yes
```

```
Boot HART ID : 0
Boot HART Domain : root
Boot HART Priv Version : unknown
Boot HART Base ISA : rv32ima
Boot HART ISA Extensions : zicntr
Boot HART PMP Count : 0
Boot HART PMP Granularity : 0
Boot HART PMP Address Bits: 0
Boot HART MHPM Count : 0
Boot HART MIDELEG : 0x00000222
Boot HART MEDELEG : 0x0000b109
[ 0.000000] Linux version 6.9.0 (lachlancomino@ubuntu-machine) (riscv32-buildroot-
   linux-gnu-gcc.br_real (Buildroot 2023.02.11-dirty) 11.4.0, GNU ld (GNU Binutils)
    2.38) #1 SMP Thu Oct 3 09:48:44 AEST 2024
[ 0.000000] Machine model: digilent_arty
[ 0.000000] SBI specification v1.0 detected
[ 0.000000] SBI implementation ID=0x1 Version=0x10003
[ 0.000000] SBI TIME extension detected
[ 0.000000] SBI IPI extension detected
[ 0.000000] SBI RFENCE extension detected
[ 0.000000] earlycon: liteuart0 at I/O port 0x0 (options '')
[ 0.000000] Malformed early option 'console'
[ 0.000000] earlycon: liteuart0 at MMIO 0xf0001000 (options '')
[ 0.000000] printk: legacy bootconsole [liteuart0] enabled
[ 0.000000] OF: reserved mem: OVERLAP DETECTED!
[ 0.000000] mmode_resv1@40f00000 (0x40f00000--0x40f40000) overlaps with
   opensbi@40f00000 (0x40f00000--0x40f80000)
[ 0.000000] OF: reserved mem: OVERLAP DETECTED!
[ 0.000000] opensbi@40f00000 (0x40f00000--0x40f80000) overlaps with
   mmode_resv0@40f40000 (0x40f40000--0x40f60000)
[ 0.000000] OF: reserved mem: 0x40f00000..0x40f3ffff (256 KiB) nomap non-reusable
   mmode_resv1@40f00000
[ 0.000000] OF: reserved mem: 0x40f00000..0x40f7ffff (512 KiB) map non-reusable
   opensbi@40f00000
[ 0.000000] OF: reserved mem: 0x40f40000..0x40f5ffff (128 KiB) nomap non-reusable
   mmode_resv0@40f40000
[ 0.000000] Zone ranges:
[ 0.000000] Normal [mem 0x0000000040000000-0x00000000ffffffff]
[ 0.000000] Movable zone start for each node
[ 0.000000] Early memory node ranges
```

```
[ 0.000000] node 0: [mem 0x0000000040000000-0x0000000040effffff]
[ 0.000000] node 0: [mem 0x0000000040f00000-0x0000000040f5ffff]
[ 0.000000] node 0: [mem 0x0000000040f60000-0x000000004fffffff]
[ 0.000000] Initmem setup node 0 [mem 0x0000000040000000-0x000000004fffffff]
[ 0.000000] SBI HSM extension detected
[ 0.000000] riscv: base ISA extensions aim
[ 0.000000] riscv: ELF capabilities aim
[ 0.000000] percpu: Embedded 11 pages/cpu s22932 r0 d22124 u45056
[ 0.000000] Kernel command line: console=liteuart earlycon=liteuart,0xf0001000
   rootwait root=/dev/ram0 ip
   =192.168.1.50:192.168.1.100:192.168.1.100:255.255.255.0::eth0:off:::
[ 0.000000] Unknown kernel command line parameters "ip
   =192.168.1.50:192.168.1.100:192.168.1.100:255.255.255.0::eth0:off:::", will be
   passed to user space.
[ 0.000000] Dentry cache hash table entries: 32768 (order: 5, 131072 bytes, linear)
[ 0.000000] Inode-cache hash table entries: 16384 (order: 4, 65536 bytes, linear)
[ 0.000000] Built 1 zonelists, mobility grouping on. Total pages: 65024
[ 0.000000] mem auto-init: stack:off, heap alloc:off, heap free:off
[ 0.000000] Memory: 189016K/262144K available (6436K kernel code, 585K rwdata, 1042K
    rodata, 242K init, 249K bss, 73128K reserved, 0K cma-reserved)
[ 0.000000] SLUB: HWalign=64, Order=0-3, MinObjects=0, CPUs=2, Nodes=1
[ 0.000000] rcu: Hierarchical RCU implementation.
[ 0.000000] rcu:  RCU restricting CPUs from NR_CPUS=32 to nr_cpu_ids=2.
[ 0.000000] rcu: RCU calculated value of scheduler-enlistment delay is 10 jiffies.
[ 0.000000] rcu: Adjusting geometry for rcu_fanout_leaf=16, nr_cpu_ids=2
[ 0.000000] NR_IRQS: 64, nr_irqs: 64, preallocated irqs: 0
[ 0.000000] riscv-intc: 32 local interrupts mapped
[ 0.000000] riscv: providing IPIs using SBI IPI extension
[ 0.000000] rcu: srcu_init: Setting srcu_struct sizes based on contention.
[ 0.000000] clocksource: riscv_clocksource: mask: 0xffffffffffffffff max_cycles: 0
   x2298375bd0, max_idle_ns: 440795208267 ns
[ 0.000020] sched_clock: 64 bits at 150MHz, resolution 6ns, wraps every
   2199023255551ns
[ 0.009873] Console: colour dummy device 80x25
[ 0.013539] Calibrating delay loop (skipped), value calculated using timer frequency
   .. 300.00 BogoMIPS (lpj=1500000)
[ 0.023955] pid_max: default: 32768 minimum: 301
[ 0.031163] Mount-cache hash table entries: 1024 (order: 0, 4096 bytes, linear)
[ 0.037671] Mountpoint-cache hash table entries: 1024 (order: 0, 4096 bytes, linear)
[ 0.076981] ASID allocator using 9 bits (512 entries)
[ 0.084125] rcu: Hierarchical SRCU implementation.
```

```
[ 0.087718] rcu:  Max phase no-delay instances is 1000.
[ 0.106195] smp: Bringing up secondary CPUs ...
[ 0.124908] smp: Brought up 1 node, 2 CPUs
[ 0.141635] devtmpfs: initialized
[ 0.202433] clocksource: jiffies: mask: 0xffffffff max_cycles: 0xffffffff,
   max_idle_ns: 19112604462750000 ns
[ 0.211624] futex hash table entries: 512 (order: 3, 32768 bytes, linear)
[ 0.240583] NET: Registered PF_NETLINK/PF_ROUTE protocol family
[ 0.254564] DMA: preallocated 128 KiB GFP_KERNEL pool for atomic allocations
[ 0.342616] cpu1: Ratio of byte access time to unaligned word access is 0.00,
   unaligned accesses are slow
[ 0.434825] cpu0: Ratio of byte access time to unaligned word access is 0.00,
   unaligned accesses are slow
[ 0.467036] platform soc: Fixed dependency cycle(s) with /soc/interrupt-
   controller@f0c00000
[ 0.482167] platform soc: Fixed dependency cycle(s) with /soc/interrupt-
   controller@f0c00000
[ 0.558727] pps_core: LinuxPPS API ver. 1 registered
[ 0.562838] pps_core: Software ver. 5.3.6 - Copyright 2005-2007 Rodolfo Giometti <
   giometti@linux.it>
[ 0.571950] PTP clock support registered
[ 0.579705] FPGA manager framework
[ 0.597305] clocksource: Switched to clocksource riscv_clocksource
[ 0.870387] NET: Registered PF_INET protocol family
[ 0.877651] IP idents hash table entries: 4096 (order: 3, 32768 bytes, linear)
[ 0.894058] tcp_listen_portaddr_hash hash table entries: 512 (order: 0, 4096 bytes,
   linear)
[ 0.901887] Table-perturb hash table entries: 65536 (order: 6, 262144 bytes, linear)
[ 0.909449] TCP established hash table entries: 2048 (order: 1, 8192 bytes, linear)
[ 0.917333] TCP bind hash table entries: 2048 (order: 3, 32768 bytes, linear)
[ 0.924211] TCP: Hash tables configured (established 2048 bind 2048)
[ 0.931350] UDP hash table entries: 256 (order: 1, 8192 bytes, linear)
[ 0.936966] UDP-Lite hash table entries: 256 (order: 1, 8192 bytes, linear)
[ 0.951892] Unpacking initramfs...
[ 0.988011] workingset: timestamp_bits=30 max_order=16 bucket_order=0
[ 1.000823] io scheduler mq-deadline registered
[ 1.004239] io scheduler kyber registered
[ 1.008804] io scheduler bfq registered
[ 1.021730] riscv-plic f0c00000.interrupt-controller: mapped 32 interrupts with 2
   handlers for 4 contexts.
[ 1.080250] No litex,nclkout entry in the dts file
```

```
[ 1.088375] LiteX SoC Controller driver initialized
[ 2.834492] f0001000.serial: ttyLXU0 at MMIO 0x0 (irq = 13, base_baud = 0) is a
   liteuart
[ 2.846821] printk: legacy console [liteuart0] enabled
[ 2.846821] printk: legacy console [liteuart0] enabled
[ 2.856681] printk: legacy bootconsole [liteuart0] disabled
[ 2.856681] printk: legacy bootconsole [liteuart0] disabled
[ 2.927478] liteeth f0002000.mac eth0: irq 14 slots: tx 16 rx 16 size 2048
[ 2.937988] i2c_dev: i2c /dev entries driver
[ 2.950607] i2c i2c-0: Not I2C compliant: can't read SCL
[ 2.954845] i2c i2c-0: Bus may be unreliable
[ 2.996491] NET: Registered PF_INET6 protocol family
[ 3.022407] Segment Routing with IPv6
[ 3.026358] In-situ OAM (IOAM) with IPv6
[ 3.030942] sit: IPv6, IPv4 and MPLS over IPv4 tunneling driver
[ 3.050433] NET: Registered PF_PACKET protocol family
[ 3.429578] clk: Disabling unused clocks
[ 4.427007] Initramfs unpacking failed: invalid magic at start of compressed archive
[ 5.128203] Freeing initrd memory: 61440K
[ 5.141095] Freeing unused kernel image (initmem) memory: 236K
[ 5.145784] Kernel memory protection not selected by kernel config.
[ 5.152867] Run /init as init process
Saving 256 bits of non-creditable seed for next boot
Starting syslogd: OK
Starting klogd: OK
Running sysctl: OK
Starting haveged: haveged: haveged: can not open UNIX socket
haveged: can not initialize command socket: Address family not supported by protocol
haveged: disabling command mode for this instance
haveged: haveged starting up
OK
Starting network: OK

Welcome to Buildroot
buildroot login: root
                  __ _
                  / / (_)__ __ ____ __
                  / /__/ / _ \/ // /\ \ /
              /____/_/_//_/\_,_//_\_\
                     / _ \/ _ \
   __ _ __ _ _\___/_//_/ ___ _
```

```
   / / (_) /____ | |/_/__| | / /____ __ / _ \(_)__ _____ __
   / /__/ / __/ -_)> </___/ |/ / -_) \ // , _/ (_-</ __/ |/ /
/____/_/\__/\__/_/|_|____|___/\__/_\_\/_/|_/_/___/\__/|___/
                  / __/ |/ / _ \
                  _\ \/ /|_/ / ___/
              /___/_/ /_/_/
   32-bit RISC-V Linux running on LiteX / VexRiscv-SMP.

login[86]: root login on 'console'
root@buildroot:~#
```

A.9 RISC-V Custom Instructions Encoding

Figure A.21: RISC-V Custom Instructions Encoding

```
// In riscv.h, the instruction is defined via:

#define opcode_R(opcode, func3, func7, rs1, rs2)   \
({                                                 \
    register unsigned long __v;                    \
    asm volatile(                                  \
        ".word ((" #opcode ") | (regnum_%0 << 7)  \
    | (regnum_%1 << 15) | (regnum_%2 << 20)    \
    | ((" #func3 ") << 12) | ((" #func7 ") << 25));"   \
        : [rd] "=r" (__v)                          \
        : "r" (rs1), "r" (rs2)        \
    );                                             \
    __v;                                           \
})

// Then the aes_custom.h envokes this instruction via the C MACROS:

#define aes_enc_round(rs1, rs2, sel)
    opcode_R(CUSTOM0, 0x00, (sel << 3), rs1, rs2)
#define aes_enc_round_last(rs1, rs2, sel)
    opcode_R(CUSTOM0, 0x00, (sel << 3) | 2, rs1, rs2)

#define aes_dec_round(rs1, rs2, sel)
    opcode_R(CUSTOM0, 0x00, (sel << 3) | 1, rs1, rs2)
#define aes_dec_round_last(rs1, rs2, sel)
    opcode_R(CUSTOM0, 0x00, (sel << 3) | 2 | 1, rs1, rs2)

// And this interface allows for OpenSSL to use the custom AES instructions
```