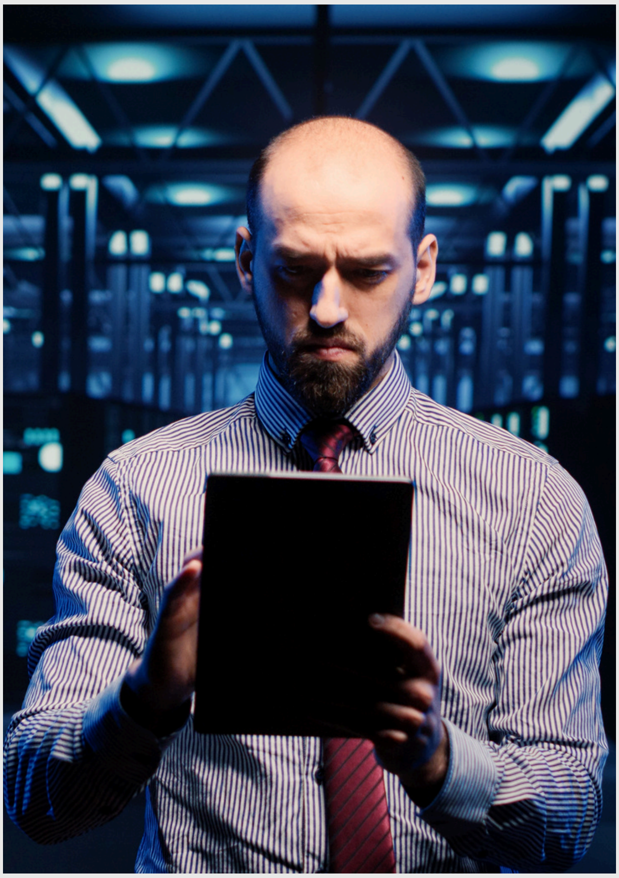
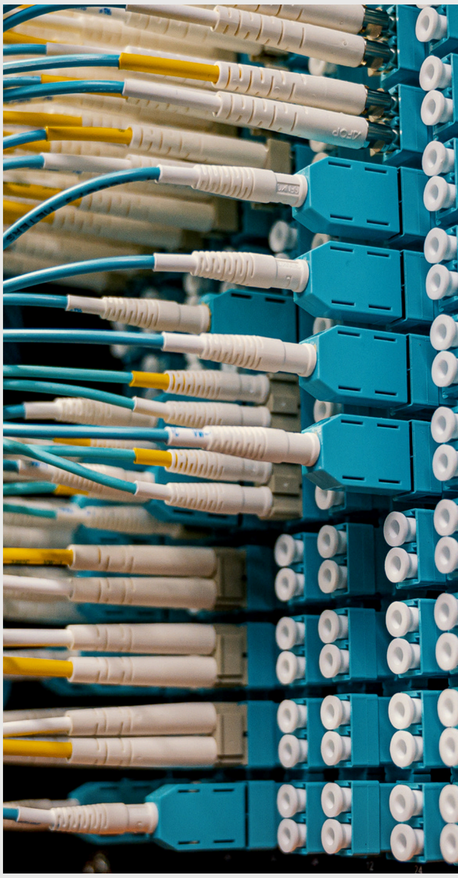
Report

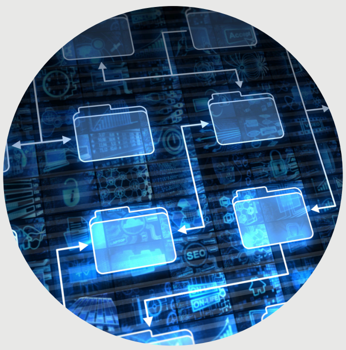




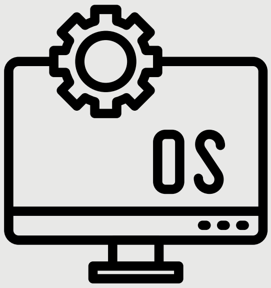
01 Project Context

As a first-year student, this project introduced me to practical low-level programming using NASM on Linux. I implemented optimized versions of core C functions from the ALSDS project, gaining hands-on experience with system-level programming, memory handling, and CPU efficiency.

By reimplementing these routines in NASM, I learned to interact directly with hardware via registers and instructions. This work helped me explore the practical differences between high-level and low-level programming and understand how performance can be improved through efficient coding at the assembly level.



02



Project Objectives

Strengthen foundational understanding of x86-64 architecture and registers.

Learn to write modular and reusable NASM code.

Replace performance-critical routines from C with optimized assembly.

Interface NASM code with C using proper conventions.

Evaluate runtime efficiency using benchmark metrics.

Apply debugging techniques to trace and fix bugs during developm

Functional Specifications

1. **initializeArray – fills array elements with a given value.**
2. **printArray – prints integer array elements using a C helper for printf.**
3. **findMax – returns the maximum value in an array.**
4. **factorial – computes the factorial of an integer.**
5. **isEven – returns 1 if the number is even, otherwise 0.**
6. **gcd – calculates the greatest common divisor of two numbers.**
7. **reverseString – reverses a null-terminated string in-place.**
8. **stringLength – returns the length of a null-terminated string.**
9. **isPalindrome – checks whether a string reads the same forwards and backwards.**

Tools and Technical Specifications

CPU Architecture: x86-64 (Intel or AMD)
Platform: Linux (Ubuntu 22.04 tested)
NASM: Assembler for x86-64
GCC: C Compiler
GDB: Debugger
Optional: QEMU or Bochs for CPU simulation (not used in my setup)

Code Strategy and Architecture

Register Usage

RDI, RSI, RDX – input parameters
RAX – used for return values and calculations
RCX, RBX – counters and temporary storage
RBP, RSP – stack management

Stack Management

All functions follow the prologue/epilogue convention:

```
push rbp
mov rbp, rsp
...
pop rbp
ret
```

Build and Execution

Makefile

```
all:
    nasm -f elf64 *.asm
    gcc -no-pie -c *.c
    gcc -no-pie -o project_exec *.o

clean:
    rm -f *.o project_exec
```

Compile and Run

```
make
./project_exec
```

Debugging and Performance Testing

GDB Commands

```
gdb project_exec
b main
run
info registers
disassemble
```

Timing with Linux time

```
time ./project_exec
```

Example result:

```
real    0m0.002s
user    0m0.001s
sys     0m0.001s
```

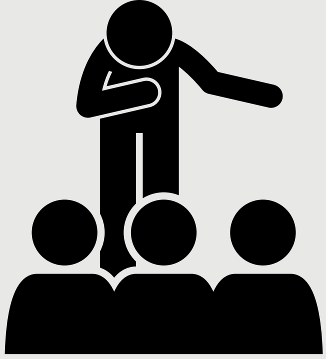
Timing with C code

```c
#include <time.h>
clock_t start = clock();
// Function calls here
clock_t end = clock();
printf("Time elapsed: %.5f seconds\n", (double)(end - s
```

Sample Output

```
Array: 7 7 7 7 42 7
Max = 42
Sum of digits of 1234 = 10
Reverse of 1234 = 4321
1234 is Even
```

Conclusion



As a student, this project gave me hands-on exposure to system programming. I gained confidence writing NASM code, managing registers, and aligning the stack. Debugging with GDB taught me to trace low-level execution and spot logic errors.

By benchmarking and optimizing NASM code, I clearly saw how assembly can outperform C in specific routines. The integration of C and NASM provided a real-world perspective on multi-language software development.

This project significantly deepened my understanding of how software and hardware interact, and helped build the foundation for more advanced systems-level courses in the future.