title: "Operating System Project Report" author: "NSCS School" date: "May 20, 2025"

# Project Team

**Team Members:**

- Belabbes Amin
- Hadj Hacene Farouk
- Benguebbour Mohammed Tedj Eddine

# Introduction

This report provides a comprehensive analysis of an operating system project that demonstrates the integration of assembly language with C programming. The project consists of various implementations of common algorithms and operations in both assembly language and C, allowing for performance comparisons and insights into low-level programming techniques.

Assembly language programming remains a crucial skill in operating system development, embedded systems, and performance-critical applications. This project showcases several key algorithms implemented in x86-64 assembly language, interfaced with C code for testing and benchmarking purposes.

## Project Overview

The project includes implementations of the following functionalities:

1. **String Operations**
2. String length calculation
3. String reversal
4. Whitespace removal
5. **Numeric Operations**
6. Bubble sort algorithm

7. Finding maximum value in an array
8. Counting even and odd numbers
9. Magic number checking
10. Perfect number verification
11. Sum of digits calculation

Each functionality is implemented in assembly language and interfaced with C code for testing and benchmarking. The project demonstrates the practical application of assembly programming in modern computing environments and provides insights into performance optimization techniques.

## Purpose and Objectives

The primary objectives of this project are:

1. To demonstrate the integration of assembly language with C programming
2. To compare the performance of assembly implementations against equivalent C implementations
3. To showcase low-level programming techniques and optimizations
4. To provide practical examples of assembly language programming for common algorithms

## Technologies Used

The project utilizes the following technologies and tools:

1. **NASM (Netwide Assembler)** - A popular assembler for the x86 architecture, used to compile assembly source code into object files.

2. **GCC (GNU Compiler Collection)** - Used to compile C source code and link object files into executable programs.

3. **x86-64 Assembly Language** - The low-level programming language used for direct hardware manipulation and optimization.

4. **C Programming Language** - Used for higher-level implementation and for interfacing with assembly code.

This combination of technologies allows for effective comparison between high-level and low-level implementations of the same algorithms, providing valuable insights into performance characteristics and optimization techniques.

# File Organization and Structure

This section provides an overview of the project files, their categorization by functionality, and the build and execution workflow.

## Project Files Overview

The project consists of multiple file types that work together to implement and test various algorithms in both assembly and C:

1. **Assembly Source Files (.asm, .s)**: These files contain the assembly language implementations of various algorithms.
2. **C Source Files (.c)**: These files contain C implementations of algorithms for comparison, as well as driver code to test and benchmark the assembly implementations.
3. **Object Files (.o)**: Compiled binary files generated from the source files, ready for linking.
4. **Executable Files**: The final compiled programs that can be run to test and benchmark the implementations.
5. **Documentation Files**: Files containing build instructions and other documentation.

## Categorization by Functionality

The project files can be categorized into the following functional groups:

### String Operations

- **String Length**: Implementation of a function to calculate the length of a string
- `stringlength.asm`, `stringlength.o`
- **String Reversal**: Implementation of a function to reverse a string in-place
- `reverse_string.asm`, `reverse_string.c`, `reverse_string.o`, `reverse_string_c.o`, `reverse_string_program`
- **Whitespace Removal**: Implementation of a function to remove whitespace from a string

- `removewhitespaces.asm`, `removewhitespaces.o`

### Numeric Operations

- **Bubble Sort**: Implementation of the bubble sort algorithm
- `bubblesort.asm`, `bubblesort.o`
- **Find Maximum**: Implementation of a function to find the maximum value in an array
- `findmax.asm`, `findmax.o`, `findmax_runner`
- **Count Even/Odd**: Implementation of a function to count even and odd numbers in an array
- `count_even_odd.asm`, `count_even_odd.c`, `count_even_odd.o`, `count_even_odd_c.o`, `count_even_odd_program`
- **Magic Number Check**: Implementation of a function to check if a number is a "magic number"
- `magic.asm`, `magic.o`, `magic_checker`
- **Perfect Number Check**: Implementation of a function to check if a number is a perfect number
- `perfect.asm`, `perfect.o`, `perfect_runner`
- **Sum of Digits**: Implementation of a function to calculate the sum of digits in a number
- `sum_of_digits.s`, `sum_of_digits.o`, `test_sum.c`, `test_sum.o`, `sum_test`

### Driver Code

- `main.c` and `main.o`: Generic driver code used with multiple assembly implementations

## File Relationships and Dependencies

The project follows a consistent pattern for each functionality:

1. An assembly source file (`.asm` or `.s`) contains the low-level implementation of an algorithm.
2. This assembly file is compiled into an object file (`.o`) using NASM.

3. A C source file (`.c`) contains either:
4. A driver program that calls the assembly function
5. A C implementation of the same algorithm for benchmarking
6. Both of the above
7. The C source file is compiled into an object file (`.o`) using GCC.
8. The object files are linked together to create an executable program.

## Build and Execution Workflow

The project uses a standard build workflow for mixed C and assembly code:

1. **Assemble the Assembly Source File**: `nasm -f elf64 -o <assembly_object>.o <assembly_source>.asm` This command compiles the assembly source file into an object file in the 64-bit ELF format.

2. **Compile the C Source File**: `gcc -c <c_source>.c -o <c_object>.o` This command compiles the C source file into an object file without linking.

3. **Link the Object Files**: `gcc -o <executable_name> <c_object>.o <assembly_object>.o` This command links the C and assembly object files to create an executable program.

4. **Run the Executable**: `./<executable_name>` This command runs the compiled program.

This workflow allows for the seamless integration of assembly and C code, enabling the development of hybrid applications that leverage the strengths of both languages.

# Assembly Implementation Analysis

This section provides a detailed analysis of the assembly implementations in the project, examining both string operations and numeric operations.

## String Operations

### String Length (stringlength.asm)

The `stringLength` function calculates the length of a null-terminated string.

```
section .text
    global stringLength

; int stringLength(char* str)
; Input:  rdi = pointer to null-terminated string
; Output: eax = length (int)

stringLength:
    push rbp
    mov rbp, rsp

    xor rcx, rcx          ; counter = 0

.loop:
    mov al, byte [rdi + rcx] ; load character at str[rcx]
    cmp al, 0              ; check for null terminator
    je .done
    inc rcx
    jmp .loop

.done:
    mov eax, ecx           ; move result to eax (return value)
    pop rbp
    ret
```

**Key features:** - Uses `rdi` register to receive the string pointer parameter (x86-64 calling convention) - Implements a simple loop that increments a counter until it finds the null terminator - Returns the count in `eax` register (return value convention for 32-bit integers) - Efficiently uses zero-extension with `xor rcx, rcx` to initialize the counter

### String Reversal (reverse_string.asm)

The `reverseString` function reverses a null-terminated string in-place.

```
section .note.GNU-stack noexec
section .text
    global reverseString

reverseString:
    ; Input:
    ; rdi = pointer to null-terminated string

    mov rsi, rdi          ; save start pointer

    ; Find string length (exclude null terminator)
```

```
    xor rcx, rcx          ; length counter = 0

.find_end:
    mov al, [rdi + rcx]
    test al, al
    jz .found_length
    inc rcx
    jmp .find_end

.found_length:
    dec rcx               ; rcx = last char index (length -1)

    ; rsi = start pointer
    ; rdi = start pointer (will be used as moving pointer)
    ; rcx = last char index

    mov rdx, 0            ; rdx = front index

.swap_loop:
    cmp rdx, rcx
    jge .done             ; done if front >= back

    ; swap characters at rsi+rdx and rsi+rcx
    mov al, [rsi + rdx]
    mov bl, [rsi + rcx]
    mov [rsi + rdx], bl
    mov [rsi + rcx], al

    inc rdx
    dec rcx
    jmp .swap_loop

.done:
    ret
```

**Key features:** - First calculates the string length to find the last character - Uses a two-pointer approach (front and back) to swap characters - Efficiently swaps characters using registers as temporary storage - Continues until the pointers meet or cross in the middle

### Whitespace Removal (removewhitespaces.asm)

The `removeWhitespaces` function removes all spaces from a string.

```
section .text
    global removeWhitespaces

; void removeWhitespaces(char* str)
```

```
; Input: rdi = pointer to string

removeWhitespaces:
    push rbp
    mov rbp, rsp

    mov rsi, rdi      ; rsi = write pointer
    mov rdx, rdi      ; rdx = read pointer

.loop:
    mov al, byte [rdx]
    cmp al, 0
    je .done

    cmp al, ' '
    je .skip

    mov [rsi], al
    inc rsi

.skip:
    inc rdx
    jmp .loop

.done:
    mov byte [rsi], 0 ; null terminate
    pop rbp
    ret
```

**Key features:** - Uses two pointers: a read pointer and a write pointer - The read pointer advances through every character - The write pointer only advances for non-space characters - Efficiently compacts the string in-place - Ensures proper null-termination of the result

## Numeric Operations

### Bubble Sort (bubblesort.asm)

The `bubbleSort` function implements the bubble sort algorithm for an integer array.

```
section .text
    global bubbleSort

; void bubbleSort(int* arr, int size)
; rdi = pointer to array
```

```
; rsi = size

bubbleSort:
    push rbp
    mov rbp, rsp

    mov rbx, rdi        ; rbx = arr pointer
    mov r8, rsi         ; r8 = size

outer_loop:
    mov rcx, 0          ; i = 0

check_outer:
    cmp rcx, r8
    jge end_outer       ; if i >= size, end

    mov rdx, 0          ; j = 0

inner_loop:
    mov r9, r8
    sub r9, rcx
    dec r9              ; r9 = size - i - 1
    cmp rdx, r9
    jge end_inner

    ; Compare arr[j] and arr[j+1]
    mov eax, [rbx + rdx*4]
    mov ecx, [rbx + rdx*4 + 4]

    cmp eax, ecx
    jle no_swap

    ; Swap
    mov [rbx + rdx*4], ecx
    mov [rbx + rdx*4 + 4], eax

no_swap:
    inc rdx
    jmp inner_loop

end_inner:
    inc rcx
    jmp check_outer

end_outer:
    pop rbp
    ret
```

**Key features:** - Implements the classic bubble sort algorithm with nested loops - Uses scaled indexing to access array elements (rdx*4 for 4-byte integers) - Optimizes the inner loop by reducing its range in each iteration - Efficiently swaps elements using registers as temporary storage

### Find Maximum (findmax.asm)

The `findMax` function finds the maximum value in an integer array.

```
section .text
    global findMax

; int findMax(int* arr, int size)
; rdi = pointer to array
; rsi = size

findMax:
    push rbp
    mov rbp, rsp

    mov rbx, rdi        ; rbx = base address of array
    mov rcx, 1          ; index = 1
    mov eax, [rbx]      ; eax = max = arr[0]

.loop:
    cmp rcx, rsi
    jge .done           ; if index >= size, exit loop

    mov edx, [rbx + rcx*4] ; edx = arr[index]
    cmp edx, eax
    jle .next           ; if arr[i] <= max, skip
    mov eax, edx        ; update max

.next:
    inc rcx
    jmp .loop

.done:
    pop rbp
    ret
```

**Key features:** - Initializes the maximum to the first element - Uses a single loop to iterate through the array - Efficiently compares each element to the current maximum - Updates the maximum only when a larger value is found

## Count Even/Odd Numbers (count_even_odd.asm)

The `countEvenOdd` function counts the number of even and odd integers in an array.

```
section .note.GNU-stack noexec
section .text
    global countEvenOdd

countEvenOdd:
    ; Arguments:
    ; rdi = pointer to int array
    ; rsi = size (number of elements)
    ; rdx = pointer to int evenCount
    ; rcx = pointer to int oddCount

    xor eax, eax          ; eax = evenCount = 0
    xor ebx, ebx          ; ebx = oddCount = 0

    test rsi, rsi
    jz done               ; if size == 0, done

    mov r8, rdi           ; r8 = current pointer to array
start
    mov r9, rsi           ; r9 = size (loop counter)

.loop:
    mov edi, [r8]         ; load current int
    test edi, 1           ; test if LSB is 1 (odd check)
    jnz .odd
    inc eax               ; evenCount++
    jmp .next

.odd:
    inc ebx               ; oddCount++

.next:
    add r8, 4             ; move pointer to next int
    dec r9
    jnz .loop

done:
    mov rdi, rdx          ; pointer to evenCount
    mov [rdi], eax        ; store evenCount

    mov rdi, rcx          ; pointer to oddCount
    mov [rdi], ebx        ; store oddCount

    ret
```

**Key features:** - Uses bit testing (`test edi, 1`) to efficiently check if a number is odd - Maintains separate counters for even and odd numbers - Optimizes loop control with a decrementing counter - Stores results through pointer parameters

### Magic Number Check (magic.asm)

The `isMagic` function checks if a number is a "magic number" (reduces to 1 when repeatedly summing its digits).

```
section .text
    global isMagic

isMagic:
    push rbp
    mov rbp, rsp
    sub rsp, 16             ; Reserve space if needed

    mov rax, rdi            ; rax = num

.top_loop:
    cmp rax, 9
    jle .check              ; If num <= 9, stop

    xor rsi, rsi            ; sum = 0

.sum_digits:
    cmp rax, 0
    je .update_sum
    xor rdx, rdx
    mov rbx, 10
    div rbx                 ; rdx = digit, rax = num / 10
    add rsi, rdx            ; sum += digit
    jmp .sum_digits

.update_sum:
    mov rax, rsi            ; num = sum
    jmp .top_loop

.check:
    cmp rax, 1
    je .magic
    mov rax, 0
    jmp .done

.magic:
    mov rax, 1

.done:
```

```
    add rsp, 16
    pop rbp
    ret
```

**Key features:** - Implements a recursive algorithm to repeatedly sum the digits - Uses division by 10 to extract individual digits - Efficiently handles the termination condition (number <= 9) - Returns a boolean result (1 for magic number, 0 otherwise)

### Perfect Number Check (perfect.asm)

The `isPerfect` function checks if a number is a perfect number (equal to the sum of its proper divisors).

```
section .text
    global isPerfect

isPerfect:
    push rbp
    mov rbp, rsp

    mov eax, edi        ; eax = input number
    cmp eax, 1
    jle .not_perfect

    mov ecx, 2          ; divisor = 2
    mov ebx, eax        ; store original number
    shr ebx, 1          ; limit = num / 2
    mov edx, 1          ; sum = 1

.loop:
    cmp ecx, ebx
    jg .done

    mov esi, eax        ; num
    xor edx, edx
    div ecx             ; divide num by ecx
    cmp edx, 0
    jne .skip

    add edx, ecx        ; add divisor to sum

.skip:
    inc ecx
    jmp .loop

.done:
```

```
    mov eax, edx
    cmp eax, edi
    sete al
    movzx eax, al
    pop rbp
    ret

.not_perfect:
    xor eax, eax
    pop rbp
    ret
```

**Key features:** - Optimizes by only checking divisors up to n/2 - Initializes sum with 1 (since 1 is always a divisor) - Uses division to check if a number is a divisor - Returns a boolean result using the SETE instruction

### Sum of Digits (sum_of_digits.s)

The `sum_of_digit` function calculates the sum of digits in a number.

```
section .note.GNU-stack noexec
section .text
global sum_of_digit   ; Export symbol for linking

sum_of_digit:
    push rbx            ; Save RBX
    mov rax, rdi        ; Get first argument (n)
    mov rcx, 10         ; Divisor for base-10 digits
    xor rbx, rbx        ; Clear sum (result)

.loop_sum:
    test rax, rax       ; Check if n == 0
    jz .fin_loop_sum_digit
    xor rdx, rdx        ; Clear upper 64 bits of dividend
(rdx:rax)
    div rcx             ; rax = quotient, rdx = remainder
(digit)
    add rbx, rdx        ; Add digit to sum
    jmp .loop_sum

.fin_loop_sum_digit:
    mov rax, rbx        ; Return sum in rax
    pop rbx             ; Restore RBX
    ret
```

**Key features:** - Uses division by 10 to extract individual digits - Accumulates the sum in the rbx register - Properly saves and restores the rbx register (callee-saved register) - Efficiently handles the loop termination condition

# C-Assembly Integration

This section examines the interface between C and assembly code, including function calling conventions, parameter passing, and memory management.

## Interface Between C and Assembly

The project demonstrates effective integration between C and assembly code through the use of external function declarations and proper adherence to calling conventions. This integration allows the project to leverage both the low-level control of assembly and the high-level abstractions of C.

### External Function Declarations

In C code, assembly functions are declared using the `extern` keyword, which informs the compiler that the function is defined elsewhere and will be resolved during linking. For example:

```
// Declaration of the assembly function
extern void countEvenOdd(int arr[], int size, int* evenCount,
int* oddCount);
```

In assembly code, functions intended to be called from C are declared as global symbols using the `global` directive:

```
section .text
    global countEvenOdd
```

This symmetrical declaration ensures that the linker can properly connect the C calls to the assembly implementations.

# Function Calling Conventions

The project follows the x86-64 System V AMD64 ABI calling convention, which defines how parameters are passed and how results are returned between functions.

## Parameter Passing

Under this convention:

1. The first six integer or pointer arguments are passed in registers:
2. RDI, RSI, RDX, RCX, R8, R9 (in that order)
3. Additional arguments are passed on the stack.
4. For floating-point arguments, XMM0-XMM7 registers are used.

Examples from the project:

```
; countEvenOdd function
; rdi = pointer to int array
; rsi = size (number of elements)
; rdx = pointer to int evenCount
; rcx = pointer to int oddCount
```

```
; bubbleSort function
; rdi = pointer to array
; rsi = size
```

## Return Values

Return values are handled as follows:

1. Integer or pointer return values are placed in RAX.
2. Floating-point return values are placed in XMM0.

Examples from the project:

```
 ; findMax function
; Return value (maximum) is placed in eax
mov eax, edx        ; update max
```

```
 ; stringLength function
; Return value (length) is placed in eax
mov eax, ecx        ; move result to eax (return value)
```

## Register Preservation

The calling convention also specifies which registers must be preserved across function calls:

1. **Caller-saved registers**: RAX, RCX, RDX, RSI, RDI, R8-R11
2. The calling function must save these if it needs their values after the call.
3. **Callee-saved registers**: RBX, RBP, RSP, R12-R15
4. The called function must preserve these registers if it uses them.

The assembly code in the project follows these conventions by:

1. Saving callee-saved registers at the beginning of functions:

   ```assembly push rbp mov rbp, rsp ```

2. Restoring them before returning:

   ```assembly pop rbp ret ```

## Stack Frame Management

Many of the assembly functions in the project use a standard stack frame setup:

```
 push rbp            ; Save the base pointer
mov rbp, rsp        ; Set up new base pointer
```

```
sub rsp, X          ; Allocate X bytes of stack space (if
needed)

; Function body

add rsp, X          ; Deallocate stack space (if allocated)
pop rbp             ; Restore the base pointer
ret                 ; Return to caller
```

This approach ensures proper stack management and facilitates debugging by maintaining a linked list of stack frames.

## Memory Management

The project demonstrates several approaches to memory management in the assembly-C interface:

### In-Place Modification

Several functions modify data in-place, using pointers passed from C:

```
 ; removeWhitespaces function
; Modifies the string in-place
mov rsi, rdi      ; rsi = write pointer
mov rdx, rdi      ; rdx = read pointer
```

### Output Parameters

Some functions use output parameters to return multiple values:

```
 ; countEvenOdd function
; Uses rdx and rcx as pointers to output variables
mov rdi, rdx          ; pointer to evenCount
mov [rdi], eax        ; store evenCount

mov rdi, rcx          ; pointer to oddCount
mov [rdi], ebx        ; store oddCount
```

### Array Handling

The project demonstrates efficient array handling in assembly:

```
; Access array elements using scaled indexing
mov eax, [rbx + rdx*4]    ; Load 32-bit integer at index rdx
```

## Integration Challenges and Solutions

### Data Type Compatibility

The project ensures data type compatibility between C and assembly by:

1. Using appropriate register sizes for different data types:
2. 32-bit integers: EAX, EBX, etc.
3. 64-bit pointers: RAX, RBX, etc.
4. Matching C data types with assembly operations:
5. `int` in C corresponds to 32-bit operations in assembly
6. `char*` in C corresponds to byte operations in assembly

### Debugging Integration Issues

The project structure facilitates debugging of integration issues by:

1. Using clear function signatures in both C and assembly
2. Maintaining consistent parameter ordering
3. Following standard calling conventions
4. Using proper stack frame setup and teardown

## Best Practices Demonstrated

The project exemplifies several best practices for C-assembly integration:

1. **Clear Documentation**: Each assembly function includes comments describing its parameters and return values.
2. **Consistent Naming**: Function names are consistent between C declarations and assembly definitions.

3. **Proper Register Usage**: Registers are used according to the calling convention.
4. **Stack Management**: Stack frames are properly set up and torn down.
5. **Register Preservation**: Callee-saved registers are properly preserved.

These practices ensure reliable integration between the C and assembly components of the project, allowing for effective performance comparison and functional verification.

# Performance Benchmarks

This section presents the methodology, results, and analysis of performance benchmarks comparing the assembly and C implementations of various functions in the project.

## Methodology

The benchmarking methodology employed in this project follows these key principles:

1. **Direct Comparison**: Each algorithm is implemented in both assembly and C to allow for direct performance comparison.
2. **Timing Measurement**: The `clock()` function from the C standard library is used to measure execution time, providing high-precision timing.
3. **Identical Inputs**: Both implementations are tested with identical inputs to ensure fair comparison.
4. **Multiple Iterations**: For functions with very short execution times, multiple iterations are performed to obtain meaningful measurements.
5. **Correctness Verification**: Results from both implementations are compared to ensure functional correctness.

Example of the benchmarking code pattern used throughout the project:

```
 // Benchmark helper function
void benchmark(void (*func)(int[], int, int*, int*), int
arr[], int size, const char* label) {
    int evenCount, oddCount;
    clock_t start = clock();
    func(arr, size, &evenCount, &oddCount);
    clock_t end = clock();
```

```
    double elapsed = (double)(end - start) / CLOCKS_PER_SEC;

    printf("%s => Even count: %d, Odd count: %d, Time: %f
seconds\n",
           label, evenCount, oddCount, elapsed);
}
```

# Results and Analysis

## Summary of Performance Comparisons

| Functionality | C Time (s) | Assembly Time (s) | Speed Ratio (C/ASM) | Winner |
|---|---|---|---|---|
| Count Even/Odd | 0.002520 | 0.001272 | 1.98 | Assembly |
| String Reversal | 0.000000 | 0.000001 | N/A* | Inconclusive** |
| Sum of Digits | 0.037360 | 0.089895 | 0.42 | C |

*The string reversal benchmark shows near-zero execution times, making the ratio calculation unreliable.*
*For string reversal, both implementations completed so quickly that meaningful comparison requires larger input or more iterations.

## Detailed Analysis by Functionality

### 1. Count Even/Odd Numbers

The assembly implementation of the count even/odd function demonstrated superior performance, completing the task approximately 1.98 times faster than the C implementation.

**Results:** - Assembly implementation: 0.001272 seconds - C implementation: 0.002520 seconds - Speed ratio (C/ASM): 1.98

**Analysis:** - The assembly version likely benefits from reduced overhead and optimized register usage - The bit testing operation (`test edi, 1`) in assembly is highly efficient for checking odd/even status - Both implementations correctly identified 500,000 even and 500,000 odd numbers, confirming functional correctness

**2. String Reversal**

Both implementations completed the string reversal task extremely quickly, with execution times close to zero, making it difficult to draw meaningful performance comparisons.

**Results:** - C implementation: 0.000000 seconds (below measurement threshold) - Assembly implementation: 0.000001 seconds (minimal measurable time)

**Analysis:** - The test string "Hello Assembly and C!" is too short to produce meaningful timing differences - Both implementations correctly reversed the string to "!C dna ylbmessA olleH" - For more meaningful comparison, larger strings or more iterations would be needed

**3. Sum of Digits**

Interestingly, the C implementation of the sum of digits function outperformed the assembly version by a factor of approximately 2.4.

**Results:** - C implementation: 0.037360 seconds - Assembly implementation: 0.089895 seconds - Speed ratio (C/ASM): 0.42 (lower is better for assembly)

**Analysis:** - Both implementations correctly calculated the sum of digits for 1234567890 as 45 - The C compiler's optimizations may have produced more efficient code than the hand-written assembly - The division operation in assembly (to extract digits) might not be optimized as well as the compiler-generated code

## Performance Insights

### 1. Assembly Advantages

Assembly code shows clear performance benefits for: - Simple, repetitive operations with minimal branching - Bit manipulation operations (as seen in the count even/odd benchmark) - Operations where register usage can be highly optimized

The count even/odd benchmark demonstrates these advantages, with the assembly implementation nearly twice as fast as the C version.

### 2. C Compiler Optimizations

Modern C compilers employ sophisticated optimization techniques that can sometimes outperform hand-written assembly code, especially for: - Complex algorithms with multiple branches - Operations involving division or floating-point calculations - Code where the compiler can apply architecture-specific optimizations

The sum of digits benchmark illustrates this point, with the C implementation outperforming the assembly version by a significant margin.

### 3. Measurement Limitations

For very fast operations, such as string reversal on short strings, standard timing methods may not provide sufficient resolution. This demonstrates the need for: - Larger input sizes - Multiple iterations - More precise timing methods

### 4. Optimization Opportunities

The benchmark results suggest several optimization opportunities: - The assembly implementation of sum of digits could be improved by optimizing the division operation - The string reversal benchmark could be enhanced with larger strings or more iterations - Additional benchmarks with varying input sizes could provide more comprehensive performance insights

## Conclusion

The performance benchmarks reveal that neither assembly nor C is universally superior. The optimal choice depends on: - The specific algorithm being implemented - The nature of the operations involved (bit manipulation, arithmetic, etc.) - The expertise of the programmer in assembly optimization - The capabilities of the C compiler's optimization

These findings highlight the value of selective use of assembly for performance-critical sections, while leveraging C for general-purpose code where compiler optimizations can be effective.

# Debugging Commands and Techniques

## Essential Debugging Commands When working with mixed C and assembly code, debugging becomes particularly important. Here are some essential debugging commands and techniques that can help identify and resolve issues: ### 1. GDB (GNU Debugger) Commands GDB is an essential tool for debugging assembly and C code:

```
 # Start GDB with your program
gdb ./your_program

# Set breakpoint at a function
break function_name
```

```
# Set breakpoint at specific address
break *0x400500

# Examine registers
info registers

# Examine specific register
p $rax

# Display memory contents
x/10xw $rsp    # Display 10 words at stack pointer

# Step through assembly instructions
stepi

# Continue execution
continue

# Display backtrace
backtrace
```

### 2. NASM Debugging Options When assembling with NASM, these options can help with debugging:

```
# Generate debug information
nasm -f elf64 -g -F dwarf your_file.asm -o your_file.o

# Generate listing file with source and machine code
nasm -f elf64 -l your_file.lst your_file.asm
```

### 3. Objdump for Disassembly Objdump can help verify the compiled assembly code:

```
# Disassemble object file
objdump -d your_file.o

# Disassemble with source code (if debug info available)
objdump -S your_file.o
```

### 4. Strace for System Call Tracing Strace can help identify issues with system calls:

```
# Trace system calls
strace ./your_program
```

## Common Debugging Scenarios and Solutions ### 1. Segmentation Faults Common causes and solutions: - **Accessing invalid memory**: Check pointer initialization and

> bounds - **Stack corruption**: Verify stack frame setup/teardown - **Register misuse**: Ensure callee-saved registers are preserved ### 2. Incorrect Results When functions return unexpected values: - **Register size mismatch**: Ensure proper register sizes for data types - **Calling convention violations**: Verify parameter passing - **Logic errors**: Step through assembly code to identify incorrect branches ### 3. Integration Issues When C and assembly don't work together: - **Symbol name mismatch**: Ensure function names match exactly - **Parameter passing errors**: Verify register usage for parameters - **Return value handling**: Check RAX/EAX for return values ## How These Commands Help These debugging commands and techniques can help: - Identify memory access violations - Trace program execution flow - Inspect register and memory state - Verify correct implementation of algorithms - Diagnose integration issues between C and assembly Proper use of these debugging tools can significantly reduce development time and help ensure correct implementation of assembly functions.

# Build and Execution Guide

This section provides detailed instructions for setting up the environment, compiling the code, and executing the programs in this project.

## Environment Setup

To work with this project, you need a Linux environment with the following tools installed:

1. **NASM (Netwide Assembler)** - For compiling assembly code `sudo apt-get install nasm`
2. **GCC (GNU Compiler Collection)** - For compiling C code and linking `sudo apt-get install gcc`

## Compilation Commands

The project uses a three-step compilation process for each program:

### 1. Assemble the Assembly Source File

```
nasm -f elf64 -o <assembly_object>.o <assembly_source>.asm
```

Parameters: - `-f elf64`: Specifies the output format for 64-bit ELF (Executable and Linkable Format) - `-o <assembly_object>.o`: Specifies the output object file - `<assembly_source>.asm`: The assembly source file to compile

Example:

```
nasm -f elf64 -o count_even_odd.o count_even_odd.asm
```

### 2. Compile the C Source File

```
gcc -c <c_source>.c -o <c_object>.o
```

Parameters: - `-c`: Compile only, do not link - `<c_source>.c`: The C source file to compile - `-o <c_object>.o`: Specifies the output object file

Example:

```
gcc -c count_even_odd.c -o count_even_odd_c.o
```

### 3. Link the Object Files

```
gcc -o <executable_name> <c_object>.o <assembly_object>.o
```

Parameters: - `-o <executable_name>`: Specifies the output executable filename - `<c_object>.o`, `<assembly_object>.o`: Object files to link

Example:

```
gcc -o count_even_odd_program count_even_odd_c.o
count_even_odd.o
```

## Execution Instructions

After successful compilation, you can run the executable:

```
./<executable_name>
```

Example:

```
./count_even_odd_program
```

## Complete Example Workflow

Here's a complete example for compiling and running the count_even_odd program:

```
# Assemble the assembly source
nasm -f elf64 -o count_even_odd.o count_even_odd.asm

# Compile the C source
gcc -c count_even_odd.c -o count_even_odd_c.o

# Link the object files
gcc -o count_even_odd_program count_even_odd_c.o
count_even_odd.o

# Run the program
./count_even_odd_program
```

## Troubleshooting

### Common Issues and Solutions

1. **"Command not found" errors**
2. Ensure NASM and GCC are installed
3. Verify that they are in your PATH
4. **Compilation errors in assembly code**
5. Check syntax for the correct NASM format
6. Ensure register usage follows x86-64 conventions
7. Verify section declarations (.text, .data, etc.)
8. **Linking errors**
9. Ensure function names match between C and assembly
10. Check that all required functions are declared as global
11. Verify object file format compatibility
12. **Runtime errors**

13. Check for proper stack management in assembly functions
14. Ensure register preservation follows calling conventions
15. Verify parameter passing between C and assembly
16. **Performance issues**
17. Use appropriate compiler optimization flags (e.g., -O2)
18. Consider using performance profiling tools
19. Review assembly code for inefficient operations

## Additional Resources

For further information on x86-64 assembly programming and C integration:

1. Intel's x86-64 Software Developer's Manual
2. System V AMD64 ABI documentation
3. NASM documentation: https://www.nasm.us/doc/
4. GCC documentation: https://gcc.gnu.org/onlinedocs/

# Conclusion

This report has examined an operating system project that demonstrates the integration of assembly language with C programming, focusing on various algorithms implemented in both languages and comparing their performance. ## Summary of Findings The project successfully implemented several key algorithms in both assembly language and C, including: 1. **String Operations** - String length calculation - String reversal - Whitespace removal 2. **Numeric Operations** - Bubble sort algorithm - Finding maximum value in an array - Counting even and odd numbers - Magic number checking - Perfect number verification - Sum of digits calculation The implementations demonstrated proper integration between C and assembly code, following standard calling conventions and memory management practices. The performance benchmarks revealed interesting insights into the relative efficiency of assembly versus C implementations. ## Performance Insights The benchmark results showed that neither assembly nor C is universally superior in terms of performance: 1. **Assembly Advantages**: The assembly implementation of the count even/odd function was approximately 1.98 times faster than the C implementation, demonstrating the efficiency of assembly for simple bit operations and tight loops. 2. **C Compiler Optimizations**: Surprisingly, the C implementation of the sum of digits function outperformed the

assembly version by a factor of approximately 2.4. This highlights the effectiveness of modern compiler optimizations for certain types of operations. 3. **Measurement Challenges**: For very fast operations like string reversal on short strings, both implementations completed so quickly that meaningful comparison was difficult, indicating the need for larger inputs or more iterations in such cases. These findings suggest that the optimal choice between assembly and C depends on the specific algorithm, the nature of the operations involved, and the expertise of the programmer in assembly optimization. ## Integration Insights The project demonstrated several key aspects of successful C-assembly integration: 1. **Calling Conventions**: Proper adherence to the x86-64 System V AMD64 ABI calling convention ensured reliable function calls between C and assembly. 2. **Parameter Passing**: Effective use of registers and memory for parameter passing facilitated seamless integration. 3. **Register Preservation**: Careful preservation of callee-saved registers maintained program stability. 4. **Stack Management**: Proper stack frame setup and teardown prevented memory corruption and facilitated debugging. These practices are essential for reliable integration of assembly code into larger C programs, especially in operating system development where stability is critical. ## Potential Improvements Based on the analysis, several potential improvements could be considered: 1. **Optimized Assembly**: Some assembly implementations, particularly the sum of digits function, could be further optimized to improve performance. 2. **Enhanced Benchmarking**: More comprehensive benchmarks with varying input sizes would provide deeper insights into performance characteristics. 3. **SIMD Instructions**: Utilizing SIMD (Single Instruction, Multiple Data) instructions could significantly improve performance for operations on arrays. 4. **Inline Assembly**: For some functions, inline assembly within C code might offer a good balance between performance and maintainability. ## Learning Outcomes This project demonstrates several important concepts in operating system development and low-level programming: 1. **Direct Hardware Interaction**: Assembly language provides direct access to hardware features, which is essential for operating system development. 2. **Performance Optimization**: The performance comparisons highlight the importance of choosing the right implementation approach for performance-critical code. 3. **Language Integration**: The successful integration of C and assembly showcases techniques for combining high-level and low-level code. 4. **Algorithm Implementation**: The various algorithms demonstrate fundamental computer science concepts implemented at a low level. In conclusion, this project provides valuable insights into the integration of assembly language with C programming, offering practical examples of low-level optimization techniques and demonstrating the trade-offs involved in choosing between high-level and low-level implementations for different algorithms.