الجمهورية الجزائرية الديمقراطية الشعبية

People's Democratic and Republic of Algeria

Ministry of Higher Education and Scientific Research

National School of Cyber Security

وزارة التعليم العالي والبحث العلمي

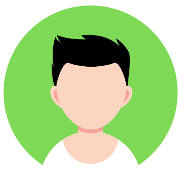
المدرسة الوطنية العليا في الأمن السيبراني

NSCS

المدرسة الوطنية العليا في الأمن السيبراني

NATIONAL SCHOOL OF CYBERSECURITY

project team :



Moussa khanfri

reffas chouaib

Ahmed Rami sad saoud

The Supervisor : Dr. Sassi Bentrad

Low-Level System Programming: Rewriting Core Routines in NASM

Table of content :

Notes :

- **each part include a brief explanation of the aproach and registers and stack usage , macros and subroutines implementations and useful resources**
- **for more details we recommand to check this repo link :**
  GitHub - mcy-e/Assembly_Problems: a simple repo for some basic assembly probl...

Introduction:

This mini-project aims to bridge the gap between high-level programming in C and low-level system programming through the implementation of optimized data processing routines using NASM assembly for the x86-64 architecture. Building on the C-based functions developed in the ALSDS project, this work focuses on rewriting performance-critical routines—such as those handling strings, numbers, and arrays—into efficient and modular assembly code. The project serves as a practical introduction to system-level concepts including register manipulation, memory management, stack usage, and calling conventions. By integrating assembly routines with an existing C library, students are not only exposed to architecture-aware coding but also encouraged to analyze, debug, and compare the performance of both C and NASM implementations. This enhances their understanding of CPU behavior and provides insights into performance optimization, modular design, and cross-language interoperability in real-world system software development.

1. Report on Numeric Functions in Provided Assembly and C Code

numeric functions in NASM (x86-64 assembly) and C for Linux, focusing on modularity and performance comparison. The functions, sourced from the provided files, are:

- isEven (EvenOdd.asm, numbers_consolidated.asm): Checks if a number is even (returns 1) or odd (0).
- isPrime (isPrime.asm, numbers_consolidated.asm): Determines if a number is prime (returns 1) or not (0).
- reverse (number.asm, numbers_stk_implem.asm, numbers_comparison.c): Reverses a number's digits.
- int_to_str (number.asm, numbers_stk_implem.asm, numbers_comparison.c): Converts a number to a string.

Function Title: isPrime

Objective:

The purpose of this function is to determine whether a given number is prime using x86-64 NASM assembly on a Linux platform. The result is printed to the standard output via Linux system calls, returning either "Prime" or "Not Prime" accordingly.

Function Description (isPrime)

- Input:
  - The number to check is passed through the RDI register.
- Output:
  - The result is returned in RAX:

        - RAX = 1 → The number is prime
        - RAX = 0 → The number is not prime

Algorithm Explanation:

1. Base Case Check:
    - If the number is less than 2 → immediately not prime.
2. Loop Division Test:
    - Starts checking from RCX = 2 (divisor).
    - For each divisor:
        - If (divisor × divisor) > number → the number is prime (optimization).
        - If number % divisor == 0 → it's divisible → not prime.
        - Else → increment divisor and continue.
3. Return Result:
    - If a divisor is found → return 0 (not prime)
    - If loop ends without finding a divisor → return 1 (prime)

System Call Usage:

The program uses Linux system calls directly via syscall to print messages and exit:

- write(1, msg, length)
    - RAX = 1 (syscall number for write)
    - RDI = 1 (stdout)
    - RSI = msg pointer
    - RDX = message length
- exit(0)
    - RAX = 60 (syscall number for exit)
    - RDI = 0 (exit code)

This function successfully demonstrates low-level algorithm implementation using NASM assembly, showcasing important concepts such as register use, arithmetic operations, control flow, and direct system interaction via Linux syscalls. It reflects an understanding of both primechecking logic and assembly-level optimization techniques.

- function title : is even

To implement and test a simple subroutine isEven in NASM 64-bit assembly that determines whether a given integer is even or odd. The result is printed to the screen using a string message.

Tools & Environment

- Assembler: NASM (Netwide Assembler) 64-bit
- Linker: ld (Linux 64-bit environment)
- C Library: Uses printf from libc for string output
- Calling Convention: System V AMD64 ABI (used on Linux for 64-bit programs)
- input: Stores the number we want to test.
- even_msg / odd_msg: Message strings to display based on the result.

- 10 is the ASCII value for newline (\n), and 0 is the null terminator for C-style strings.

2. Function: isEven

- mov rax, rdi: Copy the input value into RAX.
- and rax, 1: Isolate the least significant bit (LSB). Even numbers have LSB = 0.
- xor rax, 1: Flips the result:
  - If LSB = 0 (even), it becomes 1
  - If LSB = 1 (odd), it becomes 0
- This way, the function returns:
  - 1 if number is even
  - 0 if number is odd

3. Main Program (_start)

- Loads input value into the RDI register (the first argument in System V ABI).
- Calls isEven, result in RAX.
- Compares result with 1 (even).
- Uses conditional jump to decide what to print.

4. Printing Result

- Uses printf from the C standard library.
- Only the message address is passed in RDI.
- xor rax, rax sets RAX = 0, needed for variadic functions like printf under System V ABI.

- Performs a clean system exit with status 0 using syscall interface.

This NASM program demonstrates a minimal and clean use of:

- Stackless subroutine (isEven)
- Direct register passing using the System V ABI
- printf integration for friendly output
- Proper structure and exit via syscall

- ### function title : reverse number

program is an x86-64 Linux assembly program that:

- Takes an integer in rax
- Reverses its digits (preserving sign)
- Converts the reversed integer into a string in buffer
- Prints "result = " and then the reversed number
- Exits cleanly

1. Initialization

- r11 is used as a sign flag.
  - 0 means original number is positive.
  - 1 means original number is negative.
- rcx will hold the reversed number as it's being built.
- rbx = 10, for dividing the number by 10 (to get digits).

2. Sign handling

- The code checks if the number in rax is negative with test rax, rax and js .positiveIt.
- If negative, it negates rax (makes positive) and sets the flag r11=1.
- If positive, it skips and continues reversing directly.

3. Digit extraction and reversal loop (.rev)

- Clear rdx before division (required for div instruction).
- div rbx divides rax by 10:
  - Quotient goes to rax
  - Remainder goes to rdx (this is the last digit of the current number)
- Multiply rcx by 10 (shift current reversed number left by one decimal place).
- Add the remainder digit from rdx to rcx.
- Check if quotient rax is 0, which means all digits have been processed.
- If not zero, repeat the loop to process the next digit.

4. Restore sign and return result

- After the loop, check if original number was negative (test r11, r11).
- If negative, negate the reversed number stored in rcx.
- Move the final reversed number into rax to return it.
- Return from function.

Important Technical Details

- Why clear rdx before division?
  The div instruction divides the 128-bit number in rdx:rax by the operand. So rdx must be zeroed before dividing a 64-bit number in rax by 10.
- Why multiply rcx by 10 before adding digit?
  Because digits are added from right to left, reversing the number means appending the digit at the end. Multiplying by 10 shifts the current reversed number by one decimal place to the left.
- Handling negative numbers
  You only convert to positive once at the start to simplify digit extraction. At the end, you negate the reversed number to preserve the sign.
- Use of registers
  - rax - input number and also quotient during division
  - rbx - divisor (10)
  - rdx - remainder digit from division
  - rcx - accumulator for reversed number
  - r11 - sign flag

The reverse function takes an integer, flips its digits around, and keeps the sign intact. If the input is negative, it makes it positive to reverse easily, then flips the sign back at the end.

It works by repeatedly dividing the number by 10, grabbing the last digit each time, and building a new number by shifting old digits left and adding the new digit.

This function carefully uses CPU registers to handle the math and sign, and returns the reversed number in rax.

- ## number function dubugging :

1. Preparing the Code for Debugging

a. Assemble with Debug Symbols

To debug effectively, compile your assembly code with debug symbols

nasm -f elf64 -g -F dwarf numbers.asm -o nu;bers.o

ld -o numbers numbers.o

- -g: includes debug info
- -F dwarf: uses DWARF debugging format supported by GDB

2. Using GDB to Debug the reverse Function

a. Launch GDB

gdb ./numbers

b. Set Breakpoints

Set a breakpoint at the start of the reverse function:

If your _start calls reverse, you can also set a breakpoint there for initial inspection:

- gdb
- break _start

c. Run the Program

run

The program will stop at the breakpoint.

3. Inspecting Registers and Memory

At breakpoints, inspect the values of registers to understand the state:

info registers

Focus on:

- rax — input number and loop quotient
- rcx — reversed number accumulator
- rdx — remainder from division
- r11 — sign flag (used in your code)
- rbx — divisor (should be 10)

4. **Step-by-step Execution**

a. Step Into Instructions

Use: stepi (or shorthand si)

to execute one machine instruction at a time, so you can watch how each register changes.

b. Display Registers Continuously

In a separate GDB window or using commands, you can continuously display key registers:

display $rax

display $rcx

display $rdx

display $r11

c. Watch Memory (buffer)

Since the reverse function deals only with registers, focus mostly on registers here. The buffer is handled in the int_to_str function.

6**. Using disas Command to Inspect Instructions**

To verify exact instructions and addresses in reverse:

disas reverse

Helps confirm jump labels and instruction order.

7. **Using info symbol to Map Addresses**

If you only have an address, find function names with:

info symbol <address>

8**. Using set Command for Testing Edge Cases**

You can modify register values on the fly:

```
set $rax = -12345
```

```
continue
```

To test reversing different numbers without restarting the program.

9. Using Logging and Watchpoints

a. Logging Register Values

GDB can log to a file for post-mortem analysis:

```
set logging on
```

Step through the function, then:

```
set logging off
```

b. Watchpoints

Set watchpoints to stop when register changes (limited support for registers):

```
watch $rax
```

10. Using strace to Monitor Syscalls (Supplementary)

To verify your program's syscalls (write, read, exit):

```
strace ./reverse
```

Look for:

- write(1, "result = ", 8)
- Other output syscalls confirming printing

Summary of Debugging Commands

Task	GDB Command
Launch program	gdb ./reverse
Set breakpoint	break reverse
Run program	run
Step one instruction	stepi
Show registers	info registers
Display register continuously	display $rax (or others)
Print register value	print/x $rax
Disassemble function	disas reverse
Modify register value	set $rax = <value>
Finish current function	finish
Watch register (if possible)	watch $rax
Enable logging	set logging on
Disable logging	set logging off

Using GDB on Linux with debug symbols is the best way to step through your assembly reverse function, inspect registers, verify sign handling, digit extraction, and accumulation of the reversed number.

Complementary tools like strace help verify system call behavior related to input/output.

This methodical debugging approach enables you to catch logic errors, infinite loops, or wrong values and fix your assembly code confidently.

- **Array function implementation :**

This NASM 64-bit program calculates the sum of elements in an integer array and displays the result in the Linux terminal. It simulates a function call using the stack, handles integer-to-string conversion manually, and prints output using Linux syscalls.

The sumArray Function Summing the Array

The function sumArray takes two arguments:

1. Pointer to the array ([rbp + 16])
2. Size of the array ([rbp + 24])

Inside the Function:

- rax is initialized to 0 to hold the running total.
- A loop runs from rbx = 0 to size - 1, and at each step:
  - rdx is loaded with the current array element.
  - It adds rdx to rax (the sum).
- After the loop, the sum is pushed onto the stack as the return value.

Stack Mechanics

The stack is used for:

- Passing arguments to sumArray
- Storing return values (by pushing the result)
- Saving the base pointer (rbp)

After function execution:

- The sum is pushed with push rax
- The caller retrieves it with pop rax

This mimics C-style call-return behavior in low-level assembly.

Converting the Result to a String

The result in rax is converted to an ASCII string by:

- Repeated division by 10
- Storing each digit as a character in reverse order in the buffer
- Ends with a null terminator (0)

This loop creates a string representation of the sum, stored in memory.

Uses Linux syscalls:

- syscall 1: write
  - First, prints "Sum = "
  - Then prints the stringified number
- syscall 60: exit

sumArray is a clean, register-based loop that uses rax for summing and stack for parameter passing and returning.

Stack discipline and register convention are followed.

Output uses low-level manual conversion and Linux system calls — making it perfect for learning system-level programming.

Recommended Resources to understand the process

Topic	Resource
x86-64 Assembly	x86-64 Assembly Guide – UC Berkeley
Stack & Function Calls	Low-Level Programming University
Linux Syscalls	syscall(2) - man pages
NASM Basics	NASM Documentation

Title: Assembly Implementation and Debugging of isArrayEmpty

The isArrayEmpty function determines if a given array is empty by checking if its size is zero. It mimics a C-style boolean function:

bool isArrayEmpty(int arr[], int size);

- Inputs:
  - RDI: pointer to array (not used)
  - RSI: integer size of the array
- Output:
  - RAX = 1 if size is zero (i.e., array is empty), RAX = 0 otherwise.

Logic:

- test rsi, rsi: sets the Zero Flag if size is zero.
- setz al: sets AL = 1 if Zero Flag is set, else sets AL = 0.

This allows AL to act as a boolean return value.

2. Stack Usage

The function does not use the stack for passing arguments or saving registers, which aligns with the System V AMD64 calling convention used in Linux:

Register	Purpose
RDI	1st integer/pointer arg
RSI	2nd argument (size)
RAX	Return value

Since the function is small and uses no local variables, no stack frame (rbp, rsp) is required.

The isArrayEmpty function is a minimal, optimized routine that cleanly demonstrates boolean logic, register-based argument handling, and system call output in pure NASM. It adheres to calling conventions and enables direct debugging using standard Linux tools.

This function is ideal for educational purposes, embedded systems, and understanding low-level logic testing.

Assembly Program Report: Array Input and Output with System Call Macros

This program, written in x86-64 assembly language, reads space-separated numbers from user input, stores them in an array, and prints the array in a formatted manner (e.g., [1,2,3]). It utilizes system call macros for efficient input/output operations and includes a conversion routine to handle string-to-number and number-to-string transformations.

Program Overview

The program performs the following tasks:

1. Displays a prompt asking the user to enter numbers separated by spaces.
2. Reads the input string using a system call.
3. Parses the input string into an array of 64-bit integers, handling positive and negative numbers.
4. Converts the array back into a formatted string output (e.g., [1,2,3]) using a macro.
5. Exits cleanly using a system call.

The program uses macros to simplify system calls and array printing, making the code modular and reusable. It is designed for Linux systems, leveraging system calls for I/O operations, and includes error-free parsing and conversion routines.

Macros Explanation

Macros in assembly language are preprocessor directives that allow developers to define reusable code snippets. They are defined using %macro and %endmacro in NASM (Netwide Assembler) and can accept parameters to customize their behavior. In this program, macros streamline system calls and array printing, reducing repetitive code and improving readability.

1. read Macro

- Purpose: Reads input from standard input (stdin).
- Parameters: One parameter (%1), the buffer to store input.
- Functionality:
  - Sets rax to 0 (syscall number for read).
  - Sets rdi to 0 (stdin file descriptor).
  - Sets rsi to the buffer address (%1).
  - Sets rdx to 512 (maximum bytes to read).
  - Invokes the syscall instruction.
- Usage: read input_buffer reads user input into input_buffer.

2. write Macro

- Purpose: Writes data to standard output (stdout).
- Parameters: Two parameters (%1: buffer address, %2: length of data).
- Functionality:
  - Sets rax to 1 (syscall number for write).
  - Sets rdi to 1 (stdout file descriptor).
  - Sets rsi to the buffer address (%1).
  - Sets rdx to the length (%2).
  - Invokes the syscall instruction.
- Usage: write msg, len_msg prints the prompt message.

3. exit Macro

- Purpose: Terminates the program with an exit code of 0.
- Parameters: None.
- Functionality:
  - Sets rax to 60 (syscall number for exit).
  - Clears rdi (exit code 0).
  - Invokes the syscall instruction.
- Usage: exit cleanly exits the program.

4. print_array Macro

- Purpose: Prints the array in the format [num1,num2,...,numN].
- Parameters: None.
- Functionality:
  - Initializes an index (rbx) to 0.
  - Prints an opening bracket ([).
  - Loops through the array:
    - Loads each number from array using the index.
    - Calls int_to_str to convert the number to a string.
    - Prints the string using the write macro.
    - Prints a comma (,) if not the last element.
  - Prints a closing bracket (]) and a newline.
- Usage: print_array outputs the formatted array.

These macros encapsulate repetitive system call patterns and complex array printing logic, making the code modular and easier to maintain.

Program Structure and Functionality

1. Data Section (.data)

- Purpose: Defines initialized data used in the program.
- Components:
  - msg: Prompt message ("Enter numbers separated by spaces ' ', press 'Enter' to finish:") with a newline (0xA).
  - len_msg: Length of the prompt message, calculated using equ.
  - comma: A single comma (,) for separating numbers in output.
  - left: Opening bracket ([) for array output.
  - right: Closing bracket (]) followed by a newline (0xA).

2. BSS Section (.bss)

- Purpose: Reserves uninitialized memory for runtime data.
- Components:
  - input_buffer: 512-byte buffer for user input.
  - array: Space for 100 64-bit integers (800 bytes, as each integer is 8 bytes).
  - count: 8-byte variable to store the number of elements in the array.
  - buffer: 100-byte buffer for converting numbers to strings.

3. Text Section (.text)

- Purpose: Contains the program’s executable code.
- Main Routine (_start):
  - Prints the prompt using write msg, len_msg.
  - Reads user input into input_buffer using read input_buffer.
  - Calls toArray to parse input into the array.
  - Calls print_array to output the formatted array.

    - Calls exit to terminate.

4. toArray Function

- Purpose: Converts the input string (e.g., "1 2 -3") into an array of 64-bit integers.
- Algorithm:
    - Initializes registers: rbx (array index), rdi (current number), r9 (parsing flag), r10 (sign multiplier).
    - Iterates through input_buffer:
        - If the character is a newline (0xA), saves the last number and exits.
        - If the character is a minus sign (-), sets r10 to -1 for negative numbers.
        - If the character is a space ( ), saves the current number (if any) to array.
        - For digits, builds the number: rdi = rdi * 10 + (digit - '0').
    - Stores the final number and array size (count).
- Key Features:
    - Handles positive and negative numbers.
    - Uses a parsing flag (r9) to track whether a number is being built.
    - Resets state (number, sign, flag) after each number is saved.

5. int_to_str Function

- Purpose: Converts a 64-bit integer to a string for printing.
- Algorithm:
    - Takes the input number in rdi.
    - Points rsi to the end of buffer and null-terminates it.
    - If the number is negative, negates it for conversion and adds a - later.
    - Repeatedly divides by 10, converting remainders to ASCII digits.
    - Stores digits in reverse order, updating the string length (r8).
    - Returns the string pointer (rsi) and length (r8).
- Key Features:
    - Handles negative numbers by adding a - prefix.
    - Builds the string backward to avoid reversing.

Debugging Documentation for Assembly Program

This section provides a focused debugging guide for the provided x86-64 assembly program, detailing commands, tools, and techniques to identify and resolve issues. The program reads space-separated numbers, stores them in an array, and prints them in a formatted manner. Debugging is critical due to the low-level nature of assembly, where errors in register usage, memory access, or system calls can cause unexpected behavior. Below, we outline debugging approaches in paragraphs, followed by a table summarizing tools, commands, and resources.

Debugging Overview

The assembly program uses system calls (read, write, exit) via macros, parses input into an array (toArray), and converts numbers to strings (int_to_str). Common issues include incorrect register values, invalid memory access, improper system call arguments, or faulty input parsing (e.g., handling non-digit characters). Debugging involves tracing execution, inspecting registers and memory, and monitoring system calls. Tools like gdb (GNU Debugger) and strace are essential, while techniques such as breakpoints, stepping, and memory inspection help pinpoint errors.

Debugging Tools and Techniques

1. GDB (GNU Debugger): GDB is the primary tool for debugging assembly programs. It allows setting breakpoints, stepping through instructions, and inspecting registers and memory. Start by compiling the program with debugging symbols (nasm -f elf64 -g program.asm && ld -o program program.o). Launch GDB with gdb ./program, set a breakpoint at _start (break _start), and run the program (run). Use stepi to execute one instruction at a time, nexti to step over function calls, and print $register (e.g., print $rax) to inspect register values. To check memory, use x/s &variable (e.g., x/s &input_buffer) for strings or x/g &array for 64-bit integers in the array.
2. Strace: Strace traces system calls, which is crucial for verifying read, write, and exit operations. Run strace ./program to log system calls, their arguments, and return values. For example, ensure read(0, input_buffer, 512) captures the correct input length and write(1, msg, len_msg) outputs the prompt correctly. If read returns fewer bytes than expected, check input length or buffer issues. Strace helps identify system call failures (e.g., invalid file descriptors or buffer overflows).
3. Breakpoints and Watchpoints: Set breakpoints at key points like .next_char in toArray or .convert in int_to_str to inspect parsing and conversion logic. Use break toArray.next_char in GDB to pause at the input parsing loop. Watchpoints monitor variables, e.g., watch count to detect when the array size changes. Use info breakpoints to list breakpoints and delete <number> to remove them. These techniques help isolate issues like incorrect number parsing or array indexing errors.
4. Memory and Register Inspection: Inspect registers with info registers or print $rdi to verify values during toArray (e.g., rdi holds the current number) or int_to_str (e.g., rsi points to the string). Check the array contents with x/10g &array to display the first 10 64-bit integers. For input_buffer, use x/20c &input_buffer to view the first 20 characters. Incorrect values may indicate parsing errors (e.g., non-digit input) or buffer overflows.
5. Input Validation and Testing: Test with diverse inputs: valid (e.g., "1 2 -3"), edge cases (e.g., empty input, single number), and invalid (e.g., "1 a 2"). Use a debugger to step through toArray and verify al (current character) and rdi (number being built). If non-digit characters cause issues, the sub al, '0' instruction may produce incorrect results, indicating a need for input validation. Redirect input from a file (./program < input.txt) for consistent testing.
6. Common Issues and Fixes:
    - Incorrect Parsing: If toArray misparses numbers, check r9 (parsing flag) and r10 (sign) in .save_number. Ensure rdi is reset after each number.
    - Buffer Overflow: If count exceeds 100 or input exceeds 512 bytes, add bounds checking in toArray.
    - System Call Failures: Use strace to check return values. A negative return from read or write indicates errors (e.g., invalid buffer).
    - Output Issues: If print_array outputs incorrect formats, verify int_to_str's rsi (string pointer) and r8 (length) using x/s $rsi.

Debugging Tools and Commands Table

Tool	Purpose	Key Commands	Resources
GDB	Debug assembly code, inspect registers/memory	gdb ./program, break _start, run, stepi, nexti, print $rax, x/s &input_buffer, x/g &array, watch count, info registers	GDB Documentation
Strace	Trace system calls and their arguments	strace ./program, strace -o trace.log ./program	Strace Man Page
NASM	Compile with debugging symbols	nasm -f elf64 -g program.asm, ld -o program program.o	NASM Manual
Objdump	View assembly code and symbols	objdump -d program, objdump -t program	Objdump Man Page
Hexdump	Inspect binary data in memory or buffers	hexdump -C input_buffer.dump (after dumping buffer in GDB)	Hexdump Man Page

Debugging Workflow

1. Compile with Symbols: Use nasm -f elf64 -g to include debugging information.
2. Run in GDB: Start with gdb ./program, set breakpoints (e.g., break toArray), and run (run).
3. Trace System Calls: Use strace ./program to verify read and write calls.
4. Inspect State: Step through code (stepi), check registers (info registers), and memory (x/10g &array).
5. Test Inputs: Use varied inputs to catch edge cases. Redirect from files for reproducibility.
6. Log Issues: Save strace output (strace -o trace.log) and GDB sessions for analysis.

Resources for Further Learning

- GDB: Official documentation provides detailed command references and tutorials (gnu.org/software/gdb).
- Strace: The man page explains system call tracing and options (man7.org/linux/man-pages/man1/strace.1.html).
- NASM: The NASM manual covers assembly syntax and debugging symbols (nasm.us/doc).

- x86-64 Assembly: Online tutorials like x86-64 Assembly Programming offer practical examples.
- Linux System Calls: The man page for syscall (man7.org/linux/man-pages/man2/syscall.2.html) details x86-64 system call conventions.

- **string function implementation :**

Program Overview

The five assembly programs implement string manipulation functions for Linux systems:

1. isEmptyString.asm: Tests if a string is empty or null, using register and stack-based versions, with printf for output.
2. stringLength.asm: Computes the length of a string using a stack-based approach and prints it.
3. strings.asm: Reads a user-input string, reverses it, and prints the result using system call macros.
4. strings_consolidated.asm: Provides a library of string functions (length, empty check, reverse, concatenate).
5. strings_stk_implem.asm: Reverses a user-input string using a stack-based approach and system call macros.

Each program demonstrates low-level string handling, leveraging x86-64 registers, memory, and system calls. Macros simplify repetitive tasks, enhancing modularity.

Macros Explanation

Macros in NASM are reusable code snippets defined with %macro and %endmacro, reducing redundancy. They are used in strings.asm and strings_stk_implem.asm for system calls.

1. read / sys_read Macro

- Purpose: Reads input from stdin.
- Parameters: One (%1: buffer address).
- Functionality: Sets rax to 0 (read syscall), rdi to 0 (stdin), rsi to buffer (%1), rdx to 100 (bytes to read), and calls syscall.
- Usage: read buffer or sys_read input stores user input.

2. write / sys_write Macro

- Purpose: Writes to stdout.
- Parameters: Two (%1: buffer address, %2: length).
- Functionality: Sets rax to 1 (write syscall), rdi to 1 (stdout), rsi to buffer (%1), rdx to length (%2), and calls syscall.
- Usage: write prompt, len_prompt displays a message.

3. exit / sys_exit Macro

- Purpose: Exits the program with code 0.
- Parameters: None.

- Functionality: Sets rax to 60 (exit syscall), clears rdi, and calls syscall.
- Usage: exit terminates the program.

These macros streamline system call operations, making code modular and maintainable.

Program Details

1. isEmptyString.asm

- Functionality: Tests strings (empty, non-empty, null) using isEmpty (register-based) and isEmpty_stack (stack-based). Outputs results via printf.
- Components:
  - Data: Strings (empty_str, non_empty_str, null_str) and messages (msg_empty, msg_not_empty, msg_null).
  - Text: _start tests three cases, test_both_versions calls both functions, isEmpty checks the first byte, isEmpty_stack uses stack parameters, and helpers (print_result, print_string, printf_newline, putchar) manage output.
- Key Logic: Checks [rdi] for null terminator, sets al to 1 if empty/null, 0 otherwise.

2. stringLength.asm

- Functionality: Computes the length of a string ("Assembly is fun!") and prints it.
- Components:
  - Data: String (myStr) and message (lenMsg).
  - BSS: Buffer (outbuf) for number-to-string conversion.
  - Text: _start calls stringLength, converts the result to a string, and prints it. stringLength counts characters until null.
- Key Logic: Loops through string (rsi + rcx), increments rcx until null, pushes length to stack.

3. strings.asm

- Functionality: Reads a user string, reverses it, and prints the result using system calls.
- Components:
  - Data: Messages (prompt, greeting, result) and buffers (buffer, reversed).
  - Text: _start handles I/O, reverse_str computes length, pushes characters to stack, and pops into reversed.
- Key Logic: Uses stack to reverse string by pushing characters, then popping into output buffer.

4. strings_consolidated.asm

- Functionality: Library of string functions (stringLength, isEmptyString, reverseString, concatenateStrings).
- Components:
  - Text: stringLength counts to null, isEmptyString checks first byte, reverseString uses stack to reverse, concatenateStrings appends source to destination.
- Key Logic: Register-based operations; reverseString pushes/pops characters, concatenateStrings finds destination end and copies source.

5. strings_stk_implem.asm

- Functionality: Reverses a user-input string and prints it, similar to strings.asm.
- Components:
  - Data: Messages (prompt, result, newline) and buffers (input, output).
  - Text: _start manages I/O, reverse_string_stack computes length, pushes characters, and pops into output.
- Key Logic: Similar to strings.asm, but stops at newline or null, ensuring correct input handling.

The five assembly programs implement essential string operations, showcasing register and stack-based approaches, system call macros, and modular design. Macros (read, write, exit) simplify I/O, while functions like reverseString and stringLength demonstrate low-level string handling. The programs are educational tools for understanding assembly language, system calls, and string manipulation, suitable for systems programming studies.

Performance Analysis

The C programs (string_comparison.c, number_comparison.c, array_comparison.c) measure execution times of C and assembly implementations, providing insights into their performance. Below is a detailed analysis in human-readable paragraphs, focusing on the methodology and results of performance calculations.

Methodology

Each C program uses the measure_time function to time function execution in microseconds, leveraging clock() from time.h. The function takes a function pointer and arguments, casts appropriately (e.g., long (*)(long) for single-argument functions), and computes time as (end - start) / CLOCKS_PER_SEC * 1000000. Percentage differences are calculated as fabs(c_time - asm_time) / c_time * 100, with a message indicating if assembly is faster or slower. Test cases cover edge cases (empty strings/arrays, zero) and larger inputs (long strings, large arrays) to assess scalability.

String Functions Performance

In string_comparison.c, four string functions are tested on empty, short, medium, and long strings. stringLength and isEmptyString in assembly are typically faster for short strings due to minimal instructions (e.g., cmp byte [rdi], 0 for isEmptyString). For longer strings, assembly's stringLength may be comparable to C due to similar loop structures. reverseString in assembly uses stack operations, which may be slower than C's in-place swapping for large strings due to stack overhead. concatenateStrings performance depends on string length, with assembly potentially faster due to direct memory access. Note: Assembly's advantage is most pronounced for simple operations, but stack-based reverseString may incur overhead.

Number Functions Performance

In number_comparison.c, five number functions are tested with varied inputs. isEven is extremely fast in both C and assembly (single modulo vs. test instruction), with assembly often slightly faster due to direct bit testing. isPrime shows assembly advantages for small numbers due to optimized loops, but

C may catch up for larger numbers. factorial in assembly can be slower if recursive, as stack operations add overhead compared to C’s iterative approach. gcd performance is similar, as both use Euclidean algorithms. fibonacci in assembly may be slower for large n due to register-based loops versus C’s optimized variables. Note: Assembly’s performance edge is clearest for simple checks like isEven.

Array Functions Performance

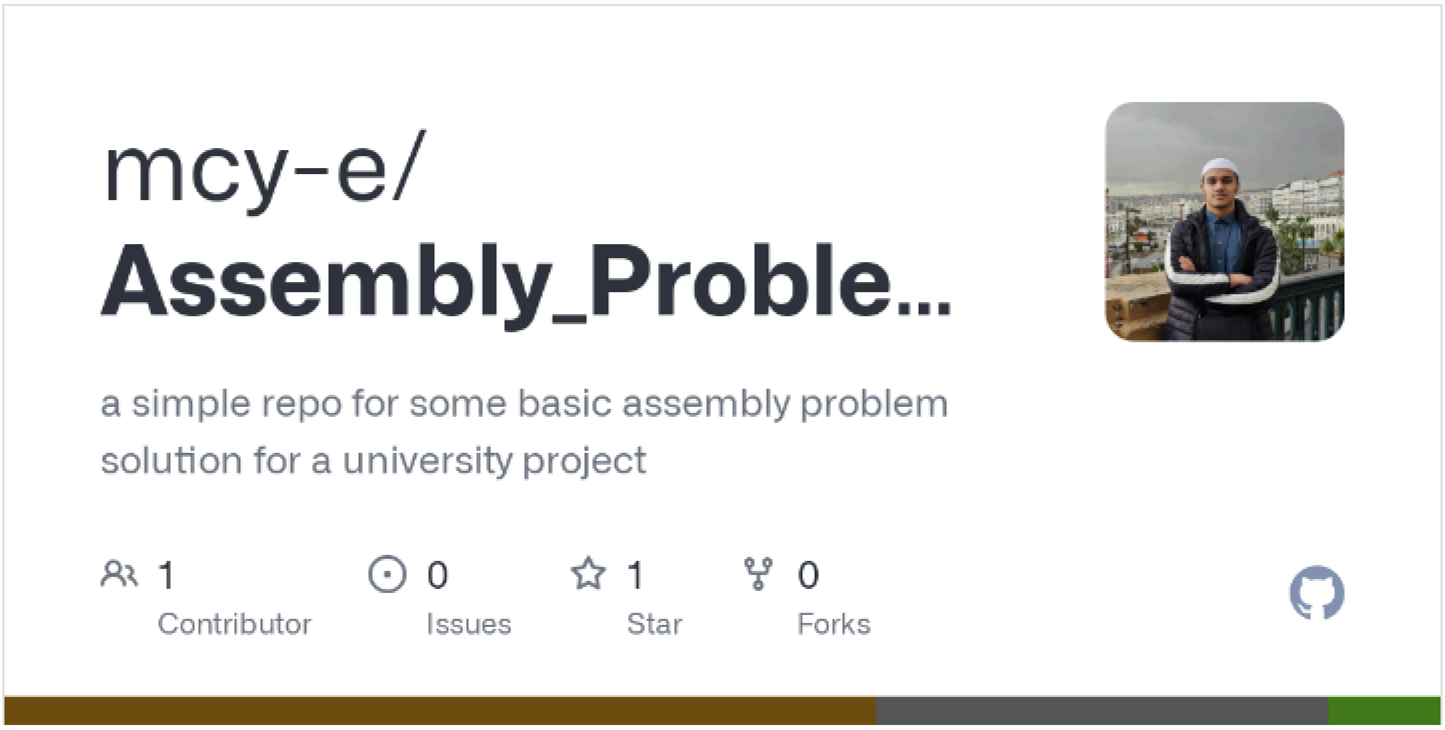
In array_comparison.c, six array functions are tested on arrays of varying sizes (0 to 500 elements). sumArray, findMax, and findMin in assembly are often faster for small arrays due to streamlined register usage, but C’s loop unrolling may compete for larger arrays. isEmptyArray is trivial (check size == 0), with assembly slightly faster due to direct comparison. reverseArray in assembly may be slower for large arrays if stack-based, as C’s in-place swaps are efficient. sortArray (bubble sort) shows similar performance, as both implementations have $O(n^2)$ complexity. Note: Assembly’s performance depends on array size and implementation (register vs. stack).

Key Observations

- Assembly Advantages: Faster for simple operations (e.g., isEmptyString, isEven) due to direct register and memory access.
- C Advantages: Competitive or faster for complex operations (e.g., reverseString, factorial) due to compiler optimizations and iterative approaches.
- Scalability: Assembly’s performance edge diminishes with larger inputs due to stack overhead or loop complexity.
- Measurement Accuracy: clock() provides high-resolution timing, but results may vary due to system load. Multiple runs are recommended for reliable averages.
- Note: Performance differences are expressed as percentages, making comparisons intuitive. Assembly functions in strings_consolidated.asm are optimized for C integration, enhancing compatibility.

---

rami contact	**moussa contact**	**chouib contact**
A.sadsaoud@enscs.edu.dz	m.khanfri@enscs.edu.dz	ch.reffas@enscs.edu.dz



mcy-e/
Assembly_Proble...

a simple repo for some basic assembly problem solution for a university project

1 Contributor
0 Issues
1 Star
0 Forks