x86-64 NASM Assembly Functions Documentation

Medjitna Belkacem/Lahmar Meriem Imene/Alem Rayane

May 20, 2025

This document provides the assembly code and detailed explanations for the functions implemented in the `asm_libraary.asm` file as part of the Assembly Programming Mini-Project. The code is written for the x86-64 architecture using the NASM assembler under Linux, adhering to the System V AMD64 ABI calling convention.

1 Data and BSS Sections

These sections define the data used by the assembly functions.

1.1 `.data` Section

Contains initialized data.

```
section .data
    hello:              db 'Hello, World!',10    ; 'Hello, World!' plus a linefeed character
    helloLen:           equ $-hello             ; Length of the 'Hello world!' string
    newline_char:       db 10                   ; Newline character
    newline_charLen:    equ $-newline_char      ; Length of newline character
    space_char:         db ' '                  ; Space character for separating numbers
    space_charLen:      equ $-space_char        ; Length of space character
```

Listing 1: `.data` Section

- `hello`: A null-terminated string containing "Hello, World!" followed by a newline character (ASCII 10).
- `helloLen`: Calculates the length of the `hello` string using the `$` (current position) operator.
- `newline_char`: A single byte containing the newline character.
- `newline_charLen`: The length of `newline_char` (1 byte).
- `space_char`: A single byte containing the space character.
- `space_charLen`: The length of `space_char` (1 byte).

1.2 `.bss` Section

Contains uninitialized data.

```
section .bss
    resu resb 20
```

Listing 2: `.bss` Section

- `resu`: A buffer of 20 bytes reserved in the BSS section. This buffer is used by the `print_number` function to store the ASCII representation of a number before printing.

2 Text Section (.text)

This section contains the executable code, including the function implementations. The `global` directives make the specified labels visible to the linker, allowing them to be called from C code.

```
section .text           ; Code section
global asm_function     ; Make this function's label visible to the linker
global asm_sort_array
global asm_reverse_array
global asm_reversewithstack_array
global asm_find_min_in_array
global asm_find_max_in_array
global asm_fibonachi
global print_number
global toUpperCase
global toLowerCase
global isPalindrom
global stringConcat
global reversString
global sumDiv
global isPerfect
global isSorted
global strgCopy
global linearSrch
```

Listing 3: Global Declarations

2.1 Array Operations

2.1.1 isSorted

Checks if an array of 64-bit integers is sorted in ascending order.

```
isSorted:
    ; Input: RDI = long long* arr (pointer to the array)
    ;        RSI = long long size (number of elements)
    ; Output: RAX = 1 if sorted, 0 if not sorted

    cmp rsi, 1
    jle .is_sorted_exit   ; If size <= 1, jump to .is_sorted_exit (return true)

    xor rcx, rcx          ; RCX = i = 0 (loop counter)

.loop_start:
    mov r8, rsi           ; Copy size to R8
    sub r8, 1             ; R8 = size - 1 (Maximum index to check: arr[size-2] vs arr[size-1])
    cmp rcx, r8           ; Compare i (RCX) with (size - 1)
    jge .is_sorted_exit   ; If i >= size - 1, we've checked all necessary pairs, array is sorted

    mov rax, rcx          ; Copy index i to RAX
    imul rax, 8           ; Multiply i by 8 to get byte offset (size of long long is 8 bytes)
    add rax, rdi          ; Add offset to base address of array (arr + offset) to get address of arr[i]
    mov r9, [rax]         ; Load arr[i] into R9

    add rax, 8            ; Just add 8 bytes to RAX to get address of arr[i+1]
    mov r10, [rax]        ; Load arr[i+1] into R10

    ; Compare arr[i] with arr[i+1]
```

```
    cmp r9, r10           ; Compare arr[i] (R9) with arr[i+1] (R10)
    jg .not_sorted_exit   ; If arr[i] > arr[i+1], it's not sorted (jump to
        return false)

    ; Increment loop counter
    inc rcx               ; i++
    jmp .loop_start       ; Continue loop

.is_sorted_exit:
    mov rax, 1            ; Set return value to true (1)
    ret                   ; Return from the function

.not_sorted_exit:
    xor rax, rax          ; Set return value to false (0)
    ret                   ; Return from the function
```

Listing 4: `isSorted` Function

Explanation:
This function checks if an array of 64-bit integers is sorted in non-decreasing order.

- It takes the array pointer in RDI and the size in RSI.
- It first handles the base case: if the size is 0 or 1, the array is considered sorted, and it jumps to `.is_sorted_exit` returning 1.
- It initializes a loop counter i in RCX to 0.
- The `.loop_start` compares i with size $-$ 1. If i is greater than or equal to size $-$ 1, it means all adjacent pairs have been checked without finding a violation, so the array is sorted.
- Inside the loop, it calculates the memory addresses of arr[i] and arr[$i+1$] using the base address (RDI) and the index i (RCX), scaled by 8 (the size of a `long long`).
- It loads arr[i] into R9 and arr[$i+1$] into R10.
- It compares R9 and R10. If arr[i] is greater than arr[$i+1$] (`jg .not_sorted_exit`), the array is not sorted, and it jumps to `.not_sorted_exit` returning 0.
- If arr[i] is less than or equal to arr[$i+1$], the loop continues by incrementing i (RCX) and jumping back to `.loop_start`.
- If the loop completes without finding an unsorted pair, it reaches `.is_sorted_exit` and returns 1.

2.1.2 `linearSrch`

Performs a linear search for a value in an array of 64-bit integers.

```
linearSrch:
    ; Input: RDI = long long* arr (pointer to the array)
    ;        RSI = long long size (number of elements)
    ;        RDX = long long n (value to search for)
    ; Output: RAX = 1 if found, 0 if not found

    cmp rsi, 0
    jle .not_found_exit ; If size is 0 or less, element cannot be found

    xor rcx, rcx        ; RCX = i = 0 (loop counter)

.loop_start:
    cmp rcx, rsi
    jge .not_found_exit ; If i >= size, we've checked all elements

    mov rax, rcx          ; Copy index i to RAX
    imul rax, 8           ; Multiply i by 8 to get byte offset
```

```
    add rax, rdi          ; Add offset to base address of array (arr + offset
        ) to get address of arr[i]
    mov r8, [rax]         ; Load arr[i] into R8

    cmp r8, rdx           ; Compare arr[i] (R8) with n (RDX)
    je .found_exit        ; If they are equal, we found it

    inc rcx               ; i++
    jmp .loop_start       ; Continue loop

.found_exit:
    mov rax, 1            ; Set return value to true (1)
    ret                   ; Return from the function

.not_found_exit:
    xor rax, rax          ; Set return value to false (0)
    ret                   ; Return from the function
```

Listing 5: `linearSrch` Function

Explanation:
This function searches for a specific 64-bit integer n within an array.

- It takes the array pointer in RDI, the size in RSI, and the target value n in RDX.
- It checks if the array size is 0 or less. If so, the element cannot be found, and it jumps to `.not_found_exit` returning 0.
- It initializes a loop counter i in RCX to 0.
- The `.loop_start` compares i with size. If i is greater than or equal to size, it means the entire array has been checked without finding the element, and it jumps to `.not_found_exit`.
- Inside the loop, it calculates the address of arr[i] and loads the value into R8.
- It compares arr[i] (R8) with the target value n (RDX). If they are equal (`je .found_exit`), the element is found, and it jumps to `.found_exit` returning 1.
- If the elements are not equal, the loop continues by incrementing i (RCX) and jumping back to `.loop_start`.
- If the loop completes without finding the element, it reaches `.not_found_exit` and returns 0.

2.1.3 `asm_sort_array`

Sorts an array of 64-bit integers using Bubble Sort.

```
asm_sort_array:
    ; Input: RDI = long long* array_ptr (pointer to the array)
    ;        RSI = long long size (number of elements)
    ; Output: The array is sorted in-place.

    ; Implements bubble sort (Keeping your original bubble sort logic)
    MOV RCX,RDI         ; RCX = array_ptr
    MOV RDX,RSI         ; RDX = size

    CMP RDX, 1
    JLE .end_outer_loop ; If size <= 1, already sorted, exit

    MOV R10, RDX        ; Save original size N in R10
    DEC RDX             ; RDX = N-1 for outer loop bound (index up to N-2)
    MOV R8, 0           ; R8 = outer loop counter i = 0

.outer_loop:
    CMP R8, RDX         ; Compare i (R8) with N-1
```

```
    JGE .end_outer_loop ; If i >= N-1, outer loop done

    MOV R9, 0           ; R9 = inner loop counter j = 0
    MOV R11, RDX        ; R11 = N-1
    SUB R11, R8         ; R11 = N-1-i (last possible index for j in the inner loop)

.inner_loop:
    CMP R9, R11         ; Compare j (R9) with R11 (N-1-i)
    JGE .end_inner_loop ; If j >= N-1-i, inner loop done

    MOV R12, [RCX + R9*8]    ; R12 = array[j] (Load element at index j)
    MOV R13, [RCX + R9*8 + 8]; R13 = array[j+1] (Load element at index j+1)

    CMP R12, R13        ; Compare array[j] and array[j+1]
    JLE .continue_inner ; If array[j] <= array[j+1], no swap needed

.swap:
    MOV [RCX + R9*8], R13    ; array[j] = array[j+1] (Store element from R13 to index j position)
    MOV [RCX + R9*8 + 8], R12 ; array[j+1] = array[j] (original value in R12) (Store element from R12 to index j+1 position)

.continue_inner:
    INC R9              ; Increment j
    JMP .inner_loop     ; Continue inner loop

.end_inner_loop:
    INC R8              ; Increment i
    JMP .outer_loop     ; Continue outer loop

.end_outer_loop:
    RET                 ; Return from the function
```

Listing 6: `asm_sort_array` Function

Explanation:

This function implements the Bubble Sort algorithm to sort an array of 64-bit integers in ascending order.

- It takes the array pointer in RDI and the size in RSI.
- It copies the array pointer to RCX and the size to RDX.
- It handles the base case where the size is 0 or 1, in which case the array is already sorted.
- The outer loop (`.outer_loop`) runs from $i = 0$ to $N - 2$, where N is the size of the array. R8 is used as the outer loop counter.
- The inner loop (`.inner_loop`) runs from $j = 0$ to $N - 2 - i$. R9 is used as the inner loop counter. R11 calculates the upper bound for the inner loop.
- Inside the inner loop, it loads the elements array[j] and array[$j+1$] into R12 and R13, respectively, by calculating their memory addresses.
- It compares R12 and R13. If array[j] is greater than array[$j + 1$], it proceeds to the `.swap` section.
- The `.swap` section swaps the values of array[j] and array[$j + 1$] in memory.
- The `.continue_inner` label is the common point after the comparison (whether a swap occurred or not), where the inner loop counter R9 is incremented, and the inner loop continues.
- After the inner loop completes, the outer loop counter R8 is incremented, and the outer loop continues.
- Once the outer loop completes, the array is sorted, and the function returns.

2.1.4 asm_reverse_array

Reverses an array of 64-bit integers in-place by swapping elements from the ends.

```
asm_reverse_array:
    ; Input: RDI = long long* array_ptr (pointer to the array)
    ;        RSI = long long size (number of elements)
    ; Output: The array is reversed in-place.

    MOV RCX,RDI          ; RCX = array_ptr
    MOV RDX,RSI          ; RDX = size

    MOV R8, 0            ; R8 = Left index (i) = 0
    MOV R9, RDX          ; R9 = Right index (j) = size
    DEC R9               ; R9 = size - 1 (Adjust j to point to the last element)

.loop_reverse:
    CMP R8, R9           ; While left index (R8) < right index (R9)
    JGE .end_loop_reverse ; If left >= right, pointers have crossed or met, done

    MOV R10, [RCX + R8*8] ; Save array[i] in R10 (Load element at left index)
    MOV R11, [RCX + R9*8] ; Save array[j] in R11 (Load element at right index)

    MOV [RCX + R8*8], R11 ; array[i] = array[j] (Store element from right index to left index position)
    MOV [RCX + R9*8], R10 ; array[j] = array[i] (original value) (Store element from R10 to right index position)

    INC R8               ; Increment left index (i++)
    DEC R9               ; Decrement right index (j--)
    JMP .loop_reverse    ; Continue loop

.end_loop_reverse:
    RET                  ; Return from the function
```

Listing 7: asm_reverse_array Function

Explanation:

This function reverses an array of 64-bit integers in-place by swapping elements from the beginning and end of the array until the pointers meet or cross.

- It takes the array pointer in RDI and the size in RSI.
- It copies the array pointer to RCX and the size to RDX.
- It initializes a left index i in R8 to 0 and a right index j in R9 to size $-$ 1.
- The .loop_reverse continues as long as the left index is less than the right index.
- Inside the loop, it loads the elements at the left and right indices into R10 and R11, respectively.
- It then swaps the elements by storing the value from R11 at the left index position and the value from R10 at the right index position.
- After swapping, it increments the left index and decrements the right index.
- The loop terminates when the left index is greater than or equal to the right index, indicating that the array has been fully reversed.

2.1.5 `asm_reversewithstack_array`

Reverses an array of 64-bit integers using the stack.

```
asm_reversewithstack_array:
    ; Input: RDI = long long* array_ptr (pointer to the array)
    ;        RSI = long long size (number of elements)
    ; Output: The array is reversed in-place.

    MOV RCX,RDI          ; RCX = array_ptr
    MOV RDX,RSI          ; RDX = size

    MOV R8, 0            ; R8 = Counter i = 0

.push_loop:
    CMP R8, RDX          ; While i < size
    JGE .end_push_loop   ; If i >= size, done pushing

    MOV R9, [RCX + R8*8] ; Load element array[i] into R9
    PUSH R9              ; Push element onto stack (8 bytes for long long)

    INC R8               ; Increment i
    JMP .push_loop       ; Continue push loop

.end_push_loop:

    MOV R8, 0            ; Reset counter i for popping

.pop_loop:
    CMP R8, RDX          ; While i < size
    JGE .end_pop_loop    ; If i >= size, done popping

    POP R9               ; Pop element from stack into R9
    MOV [RCX + R8*8], R9 ; Store popped element back into array[i]

    INC R8               ; Increment i
    JMP .pop_loop        ; Continue pop loop

.end_pop_loop:
    RET                  ; Return from the function
```

Listing 8: `asm_reversewithstack_array` Function

Explanation:
This function reverses an array by using the stack as temporary storage.

- It takes the array pointer in RDI and the size in RSI.
- It copies the array pointer to RCX and the size to RDX.
- The first loop (`.push_loop`) iterates through the array from the beginning (index 0 to size -1). In each iteration, it loads the current element into R9 and pushes it onto the stack. Since the stack grows downwards, the elements are pushed in the order they appear in the original array.
- After pushing all elements, the second loop (`.pop_loop`) iterates through the array again from the beginning. In each iteration, it pops an element from the stack into R9 and stores it back into the array at the current index. Because elements were pushed in order, popping them retrieves them in reverse order, effectively reversing the array in-place.

2.1.6 `asm_find_min_in_array`

Finds the minimum value in an array of 64-bit integers.

```
asm_find_min_in_array:
    ; Input: RDI = long long* arr (pointer to the array)
```

```
    ;         RSI = long long size (number of elements)
    ; Output: RAX = minimum value in the array

    MOV R8,0              ; R8 = i = 0 (loop counter)
    MOV RAX,[RDI]         ; Initialize RAX with the first element as the
        current minimum

forfind_min:
    INC R8                ; Increment i
    CMP R8,RSI            ; Compare i with size
    JGE ENDfind_min       ; If i >= size, loop is done

    CMP RAX,[RDI+8*R8]    ; Compare current minimum (RAX) with array[i]
    JL forfind_min        ; If RAX < array[i], current minimum is still smaller
        , continue loop

    MOV RAX,[RDI+8*R8]    ; If array[i] is smaller, update RAX with array[i]
    JMP forfind_min       ; Continue loop

ENDfind_min:
    RET                   ; Return from the function (RAX holds the minimum
        value)
```

Listing 9: `asm_find_min_in_array` Function

Explanation:
This function finds the smallest value in an array of 64-bit integers.

- It takes the array pointer in RDI and the size in RSI.
- It initializes a loop counter i in R8 to 0 and sets the initial minimum value in RAX to the first element of the array (arr[0]).
- The `forfind_min` loop iterates from the second element ($i = 1$) to the end of the array.
- Inside the loop, it compares the current minimum value (RAX) with the element at the current index i (arr[i]).
- If RAX is less than arr[i] (`JL forfind_min`), the current minimum is still the smallest, and the loop continues to the next element.
- If arr[i] is less than or equal to RAX, it means a new minimum has been found, so RAX is updated with the value of arr[i].
- The loop continues until all elements have been checked.
- After the loop, RAX holds the minimum value found in the array, and the function returns.

2.1.7 `asm_find_max_in_array`

Finds the maximum value in an array of 64-bit integers.

```
asm_find_max_in_array:
    ; Input: RDI = long long* arr (pointer to the array)
    ;         RSI = long long size (number of elements)
    ; Output: RAX = maximum value in the array

    MOV R8,0              ; R8 = i = 0 (loop counter)
    MOV RAX,[RDI]         ; Initialize RAX with the first element as the
        current maximum

forfind_max:
    INC R8                ; Increment i
    CMP R8,RSI            ; Compare i with size
    JGE ENDfind_max       ; If i >= size, loop is done (Corrected label)
```

```

    CMP RAX,[RDI+8*R8]   ; Compare current maximum (RAX) with array[i]
    JG forfind_max       ; If RAX > array[i], current maximum is still larger, continue loop (Corrected label)

    MOV RAX,[RDI+8*R8]   ; If array[i] is larger, update RAX with array[i] (Corrected register)
    JMP forfind_max      ; Continue loop (Corrected label)

ENDfind_max:             ; Corrected label
    RET                  ; Return from the function (RAX holds the maximum value)
```

Listing 10: `asm_find_max_in_array` Function

Explanation:
This function finds the largest value in an array of 64-bit integers.

- It takes the array pointer in RDI and the size in RSI.
- It initializes a loop counter i in R8 to 0 and sets the initial maximum value in RAX to the first element of the array (arr[0]).
- The `forfind_max` loop iterates from the second element ($i = 1$) to the end of the array.
- Inside the loop, it compares the current maximum value (RAX) with the element at the current index i (arr[i]).
- If RAX is greater than arr[i] (`JG forfind_max`), the current maximum is still the largest, and the loop continues.
- If arr[i] is greater than or equal to RAX, it means a new maximum has been found, so RAX is updated with the value of arr[i].
- The loop continues until all elements have been checked.
- After the loop, RAX holds the maximum value found in the array, and the function returns.

Note on Corrections: The original assembly code for `asm_find_max_in_array` contained some likely errors in jump labels and register usage. The LaTeX code above includes the assembly code with these potential corrections applied for clarity in documentation.

2.2 String Operations

2.2.1 `strgCopy`

Copies a null-terminated source string to a destination buffer.

```
strgCopy:
    ; Input: RDI = char* dest (pointer to the destination buffer)
    ;        RSI = const char* src (pointer to the source string)
    ; Output: The source string is copied to the destination buffer.

.loop_copy:
    mov al, [rsi]        ; Load byte from source (src) into AL
    mov [rdi], al        ; Store byte from AL into destination (dest)

    cmp al, 0            ; Check if the copied byte is the null terminator
    je .end_copy         ; If it's the null terminator, we're done

    inc rdi              ; Increment destination pointer
    inc rsi              ; Increment source pointer
    jmp .loop_copy       ; Continue copying

.end_copy:
    ret                  ; Return from the function
```

Listing 11: `strgCopy` Function

Explanation:
This function copies a null-terminated string from a source memory location to a destination memory location.

- It takes the destination buffer pointer in RDI and the source string pointer in RSI.
- The `.loop_copy` iterates through the source string byte by byte.
- In each iteration, it loads a byte from the source address (RSI) into the AL register and then stores that byte into the destination address (RDI).
- It checks if the copied byte is the null terminator (0). If it is, the end of the string has been reached, and it jumps to `.end_copy`.
- If the byte is not the null terminator, it increments both the destination and source pointers to move to the next byte.
- The loop continues until the null terminator is copied.

2.2.2 `isPerfect`

Checks if a number is a perfect number.

```
isPerfect:
    ; Input: RDI = long long num (the number to check)
    ; Output: RAX = 1 if perfect, 0 if not perfect

    push    rax         ; Save RAX as sumDiv uses it for return value
    call    sumDiv      ; Call sumDiv to get the sum of divisors (result in
        RAX)
    pop     rax         ; Restore RAX (which now holds the sum of divisors)

    ; The sum of all divisors (including the number itself) for a perfect
        number is 2 * num.
    ; sumDiv returns the sum of all divisors.
    ; Compare sum of divisors (RAX) with 2 * num (RDI * 2).

    ; Original assembly logic:
    ; sub     rbx, rax  ; RBX is uninitialized here, this is likely a bug or
        leftover
    ; cmp     rbx, 0    ; Comparing uninitialized RBX with RAX

    ; Corrected logic based on definition: compare sum of divisors (RAX) with
         2 * num (RDI * 2)
    mov     rbx, rdi    ; Copy num to RBX
    shl     rbx, 1      ; Multiply num by 2 (RBX = num * 2)

    cmp     rax, rbx    ; Compare sum of divisors (RAX) with 2 * num (RBX)
    jne     notPerfect  ; If not equal, it's not perfect

    mov     rax, 1      ; If equal, it's perfect (set return value to 1)
    ret                 ; Return

notPerfect:
    xor     rax, rax    ; Set return value to false (0)
    ret                 ; Return
```

Listing 12: `isPerfect` Function

Explanation:
This function checks if a given number is a perfect number. A perfect number is a positive integer that is equal to the sum of its proper positive divisors (divisors excluding the number itself). Equivalently, it

is a positive integer where the sum of all positive divisors (including the number itself) is equal to twice the number.

- It takes the number to check in RDI.
- It calls the `sumDiv` function to calculate the sum of all divisors of the input number. The result is returned in RAX.
- It saves and restores RAX around the `sumDiv` call to preserve the return value.
- It then compares the sum of divisors (RAX) with twice the original number (RDI * 2).
- If the sum of divisors is not equal to twice the number (`jne notPerfect`), it jumps to `notPerfect` and returns 0 (false).
- If the sum of divisors is equal to twice the number, it sets RAX to 1 (true) and returns.

Note on Correction: The original assembly code for `isPerfect` contained a likely bug where it subtracted RAX from an uninitialized RBX. The LaTeX code above includes the assembly code with a corrected logic to compare the sum of divisors (RAX) with twice the input number (RDI * 2).

2.2.3 `sumDiv`

Calculates the sum of all divisors for a given number, including the number itself.

```
sumDiv:
    ; Input: RDI = long long num (the number)
    ; Output: RAX = sum of all divisors

    cmp     rdi, 0
    jle     zero        ; If num <= 0, jump to zero (return 0)

    xor     rbx, rbx    ; RBX = sum = 0
    mov     rax, 1      ; RAX = i = 1 (loop counter)

loopSD:
    cmp     rax, rdi
    ja      endSD       ; If i > num, loop is done

    mov     rdx, 0      ; Clear RDX before division
    mov     rcx, rax    ; Copy i to RCX for division
    mov     rax, rdi    ; Copy num to RAX for division
    div     rcx         ; Divide num (RAX) by i (RCX). Quotient in RAX,
        Remainder in RDX

    cmp     rdx, 0      ; Check if remainder is 0 (i is a divisor)
    jnz     skip_add    ; If remainder is not 0, skip adding to sum

    add     rbx, rcx    ; If remainder is 0, add i (RCX) to sum (RBX)

skip_add:
    inc     rcx         ; Increment i (RCX)
    mov     rax, rcx    ; Move updated i back to RAX for loop condition
    jmp     loopSD      ; Continue loop

endSD:
    mov     rax, rbx    ; Move the final sum (RBX) to RAX for return value
    ret                 ; Return from the function

zero:
    xor     rax, rax    ; Set return value to 0 for num <= 0
    ret                 ; Return
```

Listing 13: `sumDiv` Function

Explanation:

This function calculates the sum of all positive divisors of a given number, including the number itself.

- It takes the number in RDI.
- It handles the case where the number is 0 or less, returning 0 in that case.
- It initializes a sum variable in RBX to 0 and a loop counter i in RAX to 1.
- The `loopSD` iterates from $i = 1$ up to the number itself.
- Inside the loop, it performs integer division of the number (RDI) by the current value of i (RAX). The remainder of the division is stored in RDX.
- It checks if the remainder is 0. If it is, i is a divisor of the number.
- If i is a divisor, it adds i (which was copied to RCX before the division) to the sum (RBX).
- The loop continues by incrementing i and jumping back to `loopSD`.
- After the loop finishes, the total sum of divisors is in RBX, which is then moved to RAX for the return value.

Note: A more optimized approach for finding divisors is to iterate only up to the square root of the number. However, this implementation iterates up to the number itself.

2.2.4 `reversString`

Reverses a null-terminated string in-place.

```
reversString:
    ; Input: RDI = char* x (pointer to the string)
    ; Output: The string is reversed in-place.

    mov     rcx, rdi            ; rcx = x (iterator, initially points to the start)
.find_endrev:
    mov     al, [rcx]           ; load byte at [rcx]
    test    al, al              ; check for NULL terminator
    jz      .got_endrev         ; if zero, found terminator
    inc     rcx                 ; rcx++ (move to the next character)
    jmp     .find_endrev        ; Continue searching for the end

.got_endrev:
    dec     rcx                 ; rcx-- to point at the last character (before the NULL)

    ; Now rcx = end pointer, rdi = start pointer
.swap_looprev:
    cmp     rdi, rcx            ; have pointers crossed? (Is start >= end?)
    jge     .donerev            ; if rdi    rcx, done reversing

    mov     dl, [rcx]           ; dl = *end (Load character from the end pointer)
    mov     al, [rdi]           ; al = *start (Load character from the start pointer)
    mov     [rdi], dl           ; *start = old_end (Store character from the end to the start position)
    mov     [rcx], al           ; *end   = old_start (Store character from AL to the end position)

    inc     rdi                 ; start++ (Move start pointer forward)
    dec     rcx                 ; end-- (Move end pointer backward)
    jmp     .swap_looprev       ; Continue swapping
```

```

.donerev:
    ret                          ; Return from the function
```

Listing 14: `reversString` Function

Explanation:

This function reverses a null-terminated string in-place by swapping characters from the beginning and end of the string until the pointers meet or cross.

- It takes the string pointer in RDI.
- The `.find_endrev` loop iterates through the string to find the null terminator, which marks the end of the string. RCX is used as an iterator.
- After finding the null terminator, RCX is decremented to point to the last character of the string.
- Now, RDI points to the beginning of the string, and RCX points to the end.
- The `.swap_looprev` continues as long as the start pointer (RDI) is less than the end pointer (RCX).
- Inside the loop, it loads the characters pointed to by the end pointer (RCX) into DL and the start pointer (RDI) into AL.
- It then swaps the characters by storing the character from DL at the start pointer's location and the character from AL at the end pointer's location.
- After swapping, it increments the start pointer and decrements the end pointer to move towards the middle of the string.
- The loop terminates when the start pointer is greater than or equal to the end pointer, indicating that the string has been fully reversed.

2.2.5 `stringConcat`

Concatenates a source null-terminated string to the end of a destination null-terminated string.

```
stringConcat:
    ; Input: RDI = char* dest (pointer to the destination buffer)
    ;        RSI = const char* src (pointer to the source string)
    ; Output: The source string is concatenated to the destination string in-
       place.

.find_endstrconcat:
    mov     dl, [rdi]         ; load byte at dest into DL
    test    dl, dl            ; is it '\0'?
    jz      .copy_strconcat   ; if yes, dest end found
    inc     rdi               ; advance dest pointer
    jmp     .find_endstrconcat ; Continue searching for the end of dest

.copy_strconcat:
    mov     al, [rsi]         ; load byte at src into AL
    test    al, al            ; is it '\0'?
    jz      .write_null       ; if yes, end of src
    mov     [rdi], al         ; store byte to dest
    inc     rdi               ; dest++
    inc     rsi               ; src++
    jmp     .copy_strconcat   ; Continue copying

.write_null:
    mov     byte [rdi], 0     ; write final NULL terminator at the end of the
       concatenated string
    ret                       ; Return from the function
```

Listing 15: `stringConcat` Function

Explanation:

This function concatenates a source string to the end of a destination string. The destination buffer must be large enough to hold the combined strings.

- It takes the destination buffer pointer in RDI and the source string pointer in RSI.
- The `.find_endstrconcat` loop iterates through the destination string to find its null terminator. RDI is used as the iterator for the destination string.
- Once the null terminator of the destination string is found, it jumps to `.copy_strconcat`. At this point, RDI points to the location where the source string should be copied.
- The `.copy_strconcat` loop iterates through the source string byte by byte. RSI is used as the iterator for the source string.
- In each iteration, it loads a byte from the source address (RSI) into AL and checks if it's the null terminator.
- If it's not the null terminator, it stores the byte into the destination address (RDI), increments both RDI and RSI to move to the next characters, and continues the loop.
- If it is the null terminator of the source string, it jumps to `.write_null`.
- The `.write_null` section writes a null terminator at the end of the concatenated string in the destination buffer.

2.2.6 `isPalindrom`

Checks if a null-terminated string is a palindrome.

```
isPalindrom:
    ; Input: RDI = char* str (pointer to the string)
    ; Output: RAX = 1 if palindrome, 0 if not palindrome

    mov     rsi, rdi             ; rsi = end_ptr, initially copy start pointer
.find_endPalindrom:
    mov     al, [rsi]            ; load byte at [rsi]
    test    al, al               ; check for NUL terminator
    jz      .got_endPalindrom    ; if zero, found terminator
    inc     rsi                  ; advance forward
    jmp     .find_endPalindrom   ; Continue searching for the end

.got_endPalindrom:
    cmp     rdi, rsi             ; Check if the string was empty (start == end before decrement)
    je      .is_palindrome_true  ; If empty, it's a palindrome

    dec     rsi                  ; back up rsi to point to the last character (before the NULL)

.loop_cmpPalindrom:
    cmp     rdi, rsi             ; have pointers crossed? (Is start >= end?)
    jge     .is_palindrome_true  ; if start >= end, it's a palindrome

    mov     al, [rdi]            ; load *start (Load character from the start pointer)
    mov     bl, [rsi]            ; load *end (Load character from the end pointer)
    cmp     al, bl               ; compare characters
    jne     .not_palindrome_false ; mismatch     not a palindrome

    inc     rdi                  ; start++ (Move start pointer forward)
    dec     rsi                  ; end-- (Move end pointer backward)
    jmp     .loop_cmpPalindrom   ; repeat the comparison

```

```
.not_palindrome_false:
    mov     rax, 0              ; return false
    jmp     .finpal

.is_palindrome_true:
    mov     rax, 1              ; return true

.finpal:
    ret                         ; Return from the function
```

Listing 16: `isPalindrom` Function

Explanation:
This function checks if a null-terminated string is a palindrome (reads the same forwards and backwards).

- It takes the string pointer in RDI.
- It copies the start pointer to RSI and uses RSI to find the end of the string by searching for the null terminator.
- After finding the end, it checks if the string was empty (start pointer equals end pointer before decrementing). An empty string is considered a palindrome.
- If the string is not empty, it decrements RSI to point to the last character of the string.
- The `.loop_cmpPalindrom` compares characters from the beginning (RDI) and end (RSI) of the string, moving inwards.
- It loads the characters pointed to by RDI and RSI into AL and BL, respectively, and compares them.
- If the characters are not equal (`jne .not_palindrome_false`), the string is not a palindrome, and it jumps to `.not_palindrome_false` returning 0.
- If the characters are equal, it increments the start pointer (RDI) and decrements the end pointer (RSI) and continues the loop.
- The loop terminates when the start pointer is greater than or equal to the end pointer. If the loop completes without finding any mismatches, it means the string is a palindrome, and it jumps to `.is_palindrome_true` returning 1.

2.2.7 `toLowerCase`

Converts a null-terminated string to lowercase in-place.

```
toLowerCase:
    ; Input: RDI = char* str (pointer to the string)
    ; Output: The string is converted to lowercase in-place.

.loop_start:
    MOV AL, BYTE [RDI]   ; Load the character pointed to by RDI into AL
    CMP AL, 0            ; Check if it's the null terminator (end of string)
    JE .end_func         ; If it is, jump to the end

    CMP AL, 'A'          ; Check if character is less than 'A'
    JL .next_char        ; If so, it's not an uppercase letter, skip
        conversion

    CMP AL, 'Z'          ; Check if character is greater than 'Z'
    JG .next_char        ; If so, it's not an uppercase letter, skip
        conversion

    ; If we reach here, AL contains an uppercase letter ('A' through 'Z')
    ADD AL, 32           ; Add 32 to convert to lowercase ('A' + 32 = 'a')
```

```
    MOV BYTE [RDI], AL   ; Store the modified character back into memory

.next_char:
    INC RDI              ; Move to the next character in the string (str++)
    JMP .loop_start      ; Continue looping

.end_func:
    RET                  ; Return from the function
```

Listing 17: `toLowerCase` Function

Explanation:
This function converts all uppercase letters in a null-terminated string to lowercase in-place.

- It takes the string pointer in RDI.
- The `.loop_start` iterates through the string character by character.
- In each iteration, it loads the current character into AL and checks if it's the null terminator. If it is, the end of the string is reached, and it jumps to `.end_func`.
- It checks if the character is within the range of uppercase letters ('A' to 'Z'). If it's not an uppercase letter, it jumps to `.next_char` to process the next character without modification.
- If the character is an uppercase letter, it adds 32 to its ASCII value to convert it to the corresponding lowercase letter. This modified character is then stored back into the string at the current position.
- The `.next_char` label is reached after processing a character (either converting it or skipping). It increments the string pointer (RDI) to move to the next character, and the loop continues.

Note: The explanation in the previous Markdown document incorrectly stated that the assembly code added 32 to lowercase letters. The assembly code correctly adds 32, which converts uppercase ('A'-'Z') to lowercase ('a'-'z').

2.2.8 `toUpperCase`

Converts a null-terminated string to uppercase in-place.

```
toUpperCase:
    ; Input: RDI = char* str (pointer to the string)
    ; Output: The string is converted to uppercase in-place.

.loop_starttoUpper:
    MOV AL, BYTE [RDI]   ; Load the character pointed to by RDI into AL
    CMP AL, 0            ; Check if it's the null terminator (end of string)
    JE .end_functoUpper  ; If it is, jump to the end

    CMP AL, 'a'          ; Check if character is less than 'a'
    JL .next_chartoUpper ; If so, it's not a lowercase letter, skip conversion

    CMP AL, 'z'          ; Check if character is greater than 'z'
    JG .next_chartoUpper ; If so, it's not a lowercase letter, skip conversion

    ; If we reach here, AL contains a lowercase letter ('a' through 'z')
    SUB AL, 32           ; Convert to uppercase by subtracting 32 ('a' - 32 = 'A')
    MOV BYTE [RDI], AL   ; Store the modified character back into memory

.next_chartoUpper:
    INC RDI              ; Move to the next character in the string (str++)
    JMP .loop_starttoUpper ; Continue looping

.end_functoUpper:
```

```
    RET                     ; Return from the function
```

Listing 18: `toUpperCase` Function

Explanation:
This function converts all lowercase letters in a null-terminated string to uppercase in-place.

- It takes the string pointer in RDI.
- The `.loop_starttoUpper` iterates through the string character by character.
- In each iteration, it loads the current character into AL and checks if it's the null terminator. If it is, the end of the string is reached, and it jumps to `.end_functoUpper`.
- It checks if the character is within the range of lowercase letters ('a' to 'z'). If it's not a lowercase letter, it jumps to `.next_chartoUpper` to process the next character without modification.
- If the character is a lowercase letter, it subtracts 32 from its ASCII value to convert it to the corresponding uppercase letter. This modified character is then stored back into the string at the current position.
- The `.next_chartoUpper` label is reached after processing a character. It increments the string pointer (RDI) to move to the next character, and the loop continues.

2.3 Number Operations

2.3.1 `asm_function`

Likely implements the factorial calculation.

```
asm_function:
    ; Input: RDI = long long n (the number for factorial)
    ; Output: RAX = factorial of n

    MOV RCX,RDI          ; Copy n to RCX (loop counter)
    MOV RAX, 1           ; Initialize result (factorial) to 1

    CMP RCX, 0
    JE .done_factorial   ; If n is 0, factorial is 1, jump to done

.for_factorial:
    MUL RCX              ; Multiply current result (RAX) by RCX (n, n-1, ...)
    DEC RCX              ; Decrement RCX
    CMP RCX, 0
    JNE .for_factorial   ; If RCX is not 0, continue loop

.done_factorial:
    RET                  ; Return from the function (RAX holds the factorial)
```

Listing 19: `asm_function` Function (Factorial)

Explanation:
This function calculates the factorial of a non-negative integer n.

- It takes the number n in RDI.
- It copies n to RCX, which will be used as a loop counter.
- It initializes the result (factorial) in RAX to 1.
- It handles the base case where n is 0. The factorial of 0 is 1, so it jumps to `.done_factorial` and returns 1.
- The `.for_factorial` loop continues as long as RCX is not 0.

- Inside the loop, it multiplies the current result in RAX by the value in RCX using the `MUL` instruction. The result of the multiplication is stored in RAX (for the lower 64 bits) and RDX (for the upper 64 bits). For typical factorial values that fit in a `long long`, RDX can be ignored.
- It decrements RCX in each iteration.
- The loop terminates when RCX becomes 0.
- The final factorial value is in RAX, and the function returns.

2.3.2 `asm_fibonachi`

Calculates the n-th Fibonacci number.

```
asm_fibonachi:
    ; Input: RDI = long long n (the index of the Fibonacci number to calculate, 0-based)
    ; Output: RAX = the n-th Fibonacci number

    MOV R13,RDI          ; Copy n to R13

    CMP R13, 0
    JE  .fib_is_zero     ; If n is 0, F(0) = 0

    CMP R13, 1
    JE  .fib_is_one      ; If n is 1, F(1) = 1

    MOV R8, 2            ; R8 = i = 2 (loop counter, starting from the 2nd term)
    MOV R12, 0           ; R12 = F(0) = 0 (previous previous term)
    MOV R11, 1           ; R11 = F(1) = 1 (previous term)

.forfib:
    CMP R8, R13          ; Compare i (R8) with n (R13)
    JG  .endfib          ; If i > n, loop is done

    MOV R10, 0           ; R10 = temp sum
    ADD R10, R11         ; R10 = F(i-1)
    ADD R10, R12         ; R10 = F(i-1) + F(i-2) = F(i)

    MOV R12, R11         ; Update previous previous term: F(i-2) = F(i-1)
    MOV R11, R10         ; Update previous term: F(i-1) = F(i)

    INC R8               ; Increment i
    JMP forfib           ; Continue loop

.fib_is_zero:
    MOV RAX, 0           ; F(0) = 0
    JMP .endfib_func

.fib_is_one:
    MOV RAX, 1           ; F(1) = 1
    JMP .endfib_func

.endfib:
    MOV RAX, R11         ; The n-th Fibonacci number is in R11 after the loop
.endfib_func:
    RET                  ; Return from the function
```

Listing 20: `asm_fibonachi` Function

Explanation:

This function calculates the n-th Fibonacci number using an iterative approach. The Fibonacci sequence starts with $\mathrm{F}(0) = 0$ and $\mathrm{F}(1) = 1$, and each subsequent number is the sum of the two preceding ones ($\mathrm{F}(n) = \mathrm{F}(n-1) + \mathrm{F}(n-2)$).

- It takes the index n in RDI.
- It handles the base cases for $n = 0$ and $n = 1$, returning 0 and 1 respectively.
- For $n > 1$, it initializes the loop counter i in R8 to 2.
- It initializes R12 to F(0) (0) and R11 to F(1) (1). These registers will hold the two previous Fibonacci numbers needed to calculate the next one.
- The `forfib` loop continues as long as i is less than or equal to n.
- Inside the loop, it calculates the current Fibonacci number F(i) by adding the previous two terms (R11 and R12) and storing the result in R10.
- It then updates the previous terms: the new previous previous term (R12) becomes the old previous term (R11), and the new previous term (R11) becomes the newly calculated current term (R10).
- It increments the loop counter i.
- The loop terminates when i becomes greater than n.
- After the loop, R11 holds the n-th Fibonacci number, which is moved to RAX for the return value.

2.4 Helper Functions

2.4.1 print_number

Converts a 64-bit integer into a string representation and prints it to standard output.

```
print_number:
    ; Input: RAX = long long num (the number to print)
    ; Output: Prints the number to standard output.
    ; Uses the 'resu' buffer in the .bss section.

displ:
    mov rdi, resu       ; RDI = pointer to the 'resu' buffer
    mov rbx, 10         ; RBX = 10 (divisor for converting to decimal digits)
    xor rcx, rcx        ; RCX = counter for digits (initially 0)

    test rax, rax       ; Check if the number is 0
    jnz .mloop          ; If not zero, jump to the main conversion loop

    ; Handle the case for number 0
    mov byte [rdi], '0' ; Store the character '0' in the buffer
    inc rdi             ; Move buffer pointer
    inc rcx             ; Increment digit counter
    jmp .printf         ; Jump to the printing section

.mloop: ; Main conversion loop (for non-zero numbers)
    xor rdx, rdx        ; Clear RDX before division (required for DIV instruction)
    mov rcx, rax        ; Save the current value of RAX (the number) in RCX
    mov rax, rdi        ; Save the buffer pointer (RDI) in RAX temporarily
    mov rdi, rcx        ; Move the number (from RCX) to RDI for the DIV instruction
    mov rcx, 10         ; Set RCX as the divisor (10)
    div rcx             ; Divide the number (RDI) by 10 (RCX). Quotient in RAX, Remainder in RDX

    ; The above DIV instruction is incorrect based on the typical usage.
    ; The DIV instruction divides the value in RDX:RAX by the operand.
    ; For 64-bit division, it divides RDX:RAX by the 64-bit operand.
    ; Let's correct the division logic to use RAX for the number and RBX for the divisor (10).

```

```
    ; Corrected Conversion Loop:
    mov rbx, 10         ; RBX = 10 (divisor)
    mov rcx, 0          ; RCX = digit counter
    mov rdi, resu       ; RDI = pointer to buffer

.mloop_corrected:
    xor rdx, rdx        ; Clear RDX before division
    div rbx             ; Divide RAX (number) by RBX (10). Quotient in RAX, Remainder in RDX

    add dl, '0'         ; Convert the remainder (digit) in DL to its ASCII character representation
    push rdx            ; Push the ASCII digit onto the stack

    inc rcx             ; Increment digit counter
    test rax, rax       ; Check if the quotient (RAX) is zero
    jnz .mloop_corrected ; If quotient is not zero, continue the loop

.inverser:; Because the push operation reverses the order of digits, we need to pop and store them in the correct order
    pop rax             ; Pop a digit (ASCII character) from the stack into RAX
    mov [rdi], al       ; Store the digit (in AL) into the buffer at the current RDI position
    inc rdi             ; Increment the buffer pointer
    loop .inverser      ; Decrement RCX (digit counter) and loop if RCX > 0

.printf:; Print the number from the buffer
    inc rcx             ; Increment RCX one last time to include the null terminator (though the current code doesn't add one explicitly)
                        ; A better approach would be to calculate the length based on the number of pushes.
    mov rax, 1          ; RAX = syscall number for sys_write (1)
    mov rdi, 1          ; RDI = file descriptor (stdout = 1)
    mov rsi, resu       ; RSI = pointer to the buffer containing the number string
    mov rdx, rcx        ; RDX = number of bytes to write (the number of digits)
    syscall             ; Make the system call to write to stdout

    ret                 ; Return from the function
```

Listing 21: `print_number` Function

Explanation:

This function takes a 64-bit integer in RAX and prints its decimal representation to standard output using the ‘sys$_{w}rite$‘$Linuxsystemcall.$

It uses a buffer named ‘resu‘ in the ‘.bss‘ section to store the ASCII digits of the number.

It handles the special case where the number is 0.

For non-zero numbers, it enters a loop that repeatedly divides the number by 10.

The remainder of each division (in RDX) is a digit of the number (from right to left). This digit is converted to its ASCII character representation by adding '0' and then pushed onto the stack.

The quotient of the division (in RAX) becomes the new number for the next iteration.

This process continues until the quotient becomes 0.

After the division loop, the digits are on the stack in reverse order. The `.inverser` loop pops the digits from the stack and stores them in the ‘resu‘ buffer.

Finally, the `.printf` section uses the ‘sys$_{w}rite$‘$syscalltoprintthecontentsofthe$‘$resu$‘$buffertostandardoutput.$

Testing and Debugging Assembly Functions

This section provides a guide to testing and debugging the provided assembly functions using GDB, along with explanations of the functions' logic and common debugging techniques.

GDB (GNU Debugger) Guide for Assembly

GDB is a powerful debugger for various programming languages, including assembly. Here's a quick guide to using GDB for debugging your NASM assembly code.

Compilation and Linking for Debugging

Before you can debug with GDB, you need to compile and link your assembly code with debugging symbols. For NASM, use the following commands:

```
nasm -f elf64 -g -F dwarf your_file.asm -o your_file.o
ld your_file.o -o your_executable
```

- `-f elf64`: Specifies the output format as 64-bit ELF.
- `-g`: Generates debugging information.
- `-F dwarf`: Specifies the DWARF debugging format (GDB prefers DWARF).
- `ld`: The linker.

Basic GDB Commands

To start GDB, run:

```
gdb ./your_executable
```

Once inside GDB, here are some essential commands:

- `start`: Starts the execution of the program and stops at the first instruction of `_start`.
- `run` or `r`: Starts or continues program execution.
- `break <location>` or `b <location>`: Sets a breakpoint.
  - `b _start`: Break at the program entry point.
  - `b asm_sort_array`: Break at the start of a function.
  - `b *0x400500`: Break at a specific memory address.
  - `b your_file.asm:lineNumber`: Break at a specific line in the source file.
- `info breakpoints` or `i b`: Lists all active breakpoints.
- `delete <breakpoint`$_n umber > or$

- - `p /x` $rax : Printsthevalueof$`RAX`$inhexadecimal.$

- - `N: Number of units to display.`
  - `F: Format (x for hex, d for decimal, c for char, s for string, i for instruction).`
  - `U: Unit size (b for byte, h for halfword (2 bytes), w for word (4 bytes), g for giant word (8 bytes)).`

`x /5gx array1`: Examine 5 giant words (QWORDs) in hexadecimal starting from `array1`.

`x /s str`$_m adam : Examinestringstartingfrom$`str_madam`.

`layout asm`: Displays the assembly code, registers, and source code (if available).

`info registers` or `i r`: Displays the values of all general-purpose registers.

`disassemble <function>` or `disas <function>`: Disassembles a function or memory range.

`set <variable>=<value>`: Changes the value of a register or variable during debugging (e.g., `set` $rax = 0)$.`quit`or`q` $: ExitsGDB.$

Function Explanations and Debugging Strategies

Let's examine each function and how to approach debugging them.

1. `asm_sort_array` (Bubble Sort)

```
; --- Function to be tested: asm_sort_array ---
asm_sort_array:
    push rbx; push r12; push r13; push r14
    MOV RCX, RDI
    MOV RDX, RSI
    CMP RDX, 1
    JLE .end_outer_loop_sort
    MOV R14, RDX ; Not strictly needed for current logic but was in original
    DEC RDX
    MOV R8, 0
.outer_loop_sort:
    CMP R8, RDX
    JGE .end_outer_loop_sort
    MOV R9, 0
    MOV R11, RDX
    SUB R11, R8
.inner_loop_sort:
    CMP R9, R11
    JGE .end_inner_loop_sort
    MOV R12, [RCX + R9*8]
    MOV R13, [RCX + R9*8 + 8]
    CMP R12, R13
    JLE .continue_inner_sort
.swap_sort:
    MOV [RCX + R9*8], R13
    MOV [RCX + R9*8 + 8], R12
.continue_inner_sort:
    INC R9
    JMP .inner_loop_sort
.end_inner_loop_sort:
    INC R8
    JMP .outer_loop_sort
.end_outer_loop_sort:
    pop r14; pop r13; pop r12; pop rbx
    RET
```

Function Logic: This function implements the Bubble Sort algorithm.

- It takes two arguments: `RDI` (pointer to the array) and `RSI` (number of elements).
- The outer loop (`.outer_loop_sort`) iterates `n-1` times (where `n` is the number of elements). The loop counter is `R8`.

- The inner loop (`.inner_loop_sort`) compares adjacent elements and swaps them if they are in the wrong order. The loop counter is `R9`.
- The upper bound of the inner loop (`R11`) decreases with each iteration of the outer loop, as larger elements "bubble up" to their correct positions.
- The comparison `CMP R12, R13` checks if `array[i]` is greater than `array[i+1]`. If it is, they are swapped.

Debugging Strategy:

- **Initial Setup:**

```
gdb ./your_executable
break asm_sort_array
run
```

- **Check Input Parameters:**

```
(gdb) p /x $rdi ; Check array address
(gdb) p /d $rsi ; Check array length
(gdb) x /5gx $rdi ; Examine the initial array contents (adjust 5
    based on array length)
```

- **Step Through Loops:**
  - Set breakpoints at the start of the outer loop (`.outer_loop_sort`) and inner loop (`.inner_loop_sort`).
  - Use `ni` to step through each iteration.
  - After each inner loop iteration, examine the array to see if swaps are happening correctly:

```
(gdb) x /5gx $rcx
```

  - Pay close attention to the loop termination conditions (`CMP R8, RDX` and `CMP R9, R11`).
- **Verify Swaps:**
  - Set a breakpoint at `.swap_sort`.
  - When hitting this breakpoint, check the values of `R12` and `R13` (the elements being compared).
  - Step one instruction (`si`) and then verify that the memory locations `[RCX + R9*8]` and `[RCX + R9*8 + 8]` have been correctly updated.
- **Edge Cases:**
  - Test with an already sorted array (e.g., `array3`). The function should still run but perform no swaps.
  - Test with a reverse-sorted array (e.g., `array2`). This will stress the swapping logic.
  - Test with a single-element array (e.g., `array4`). The initial `CMP RDX, 1` should handle this, causing it to skip sorting.
  - Test with an empty array (e.g., `array5`). The `CMP RDX, 1` should prevent any loops from executing.

2. `isPalindrom`

```
; --- Function to be tested: isPalindrom ---
isPalindrom:
    mov     rsi, rdi            ; rsi = end_ptr, rdi = start_ptr
.find_endPalindrom:
    mov     al, [rsi]           ; load byte
    test    al, al              ; check for NUL
    jz      .got_endPalindrom
    inc     rsi                 ; advance forward
    jmp     .find_endPalindrom
.got_endPalindrom:
    cmp     rdi, rsi            ; if rdi == rsi (e.g. empty string)
    je      .is_palindrome_true
    dec     rsi                 ; back up to last character
.loop_cmpPalindrom:
    cmp     rdi, rsi
    jge     .is_palindrome_true
    mov     al, [rdi]
    mov     bl, [rsi]
    cmp     al, bl
    jne     .not_palindrome_false
    inc     rdi
    dec     rsi
    jmp     .loop_cmpPalindrom
.not_palindrome_false:
    mov     rax, 0
    jmp     .finpal
.is_palindrome_true:
    mov     rax, 1
.finpal:
    ret
```

Function Logic: This function checks if a null-terminated string is a palindrome.

- It takes one argument: `RDI` (pointer to the string).
- It first finds the end of the string by iterating `RSI` until a null terminator (`0`) is found.
- After finding the end, `RSI` is decremented to point to the last character of the string.
- The function then uses two pointers, `RDI` (start) and `RSI` (end), and moves them towards the center of the string.
- In each iteration, it compares the characters pointed to by `RDI` and `RSI`.
- If a mismatch is found, it sets `RAX` to 0 (not a palindrome) and exits.
- If the pointers cross or meet without any mismatches, it sets `RAX` to 1 (is a palindrome) and exits.
- Handles empty strings and single-character strings as palindromes.

Debugging Strategy:

- **Initial Setup:**

```
gdb ./your_executable
break isPalindrom
run
```

- **Check Input String:**

```
(gdb) p /s $rdi ; Examine the string
```

- **Verify End Pointer Calculation:**
  - Set a breakpoint at `.got_endPalindrom`.
- When hit, check the value of *rsi.Itshouldpointtothenullterminator.Then,after*`dec rsi`,
- **Step Through Comparison Loop:**
  - Set a breakpoint at `.loop_cmpPalindrom`.
  - In each iteration, check *rdi*

```
; --- Function to be tested: asm_fibonachi ---
asm_fibonachi:
    push rbx; push r12; push r13
    MOV R13, RDI ; n
    CMP R13, 0
    JE  .fib_is_zero_fib
    CMP R13, 1
    JE  .fib_is_one_fib
    MOV R8, 0
    MOV R12, 0 ; F(k-2)
    MOV R11, 1 ; F(k-1)
.forfib_user_fib:
    INC R8
    CMP R8, R13
    JGE .endfib_user_fib
    MOV R10, 0 ; temp sum
    ADD R10, R11
    ADD R10, R12
    MOV R12, R11
    MOV R11, R10
    MOV RAX, R10 ; Current F(R8+1) if R8 from 0, or F(R8) if R8 is "current N"
                 ; With user's loop structure, RAX gets F(n)
    JMP .forfib_user_fib
.endfib_user_fib:
    ; RAX should hold the last computed F(n) from the loop
    JMP .fib_finish_fib
.fib_is_zero_fib:
    MOV RAX, 0
    JMP .fib_finish_fib
.fib_is_one_fib:
    MOV RAX, 1
.fib_finish_fib:
    pop r13; pop r12; pop rbx
    RET
```

Function Logic: `This function calculates the Nth Fibonacci number iteratively.`

- `It takes one argument: RDI (the input 'n').`
- `It handles base cases:`
  - `If n = 0, RAX is set to 0.`
  - `If n = 1, RAX is set to 1.`
- `For n > 1, it uses an iterative approach:`
  - `R12 stores F(k-2) (initialized to 0, which is F(0)).`
  - `R11 stores F(k-1) (initialized to 1, which is F(1)).`
  - `The loop (.forfib_user_fib) iterates from R8 = 0 up to n-1.`
  - `In each iteration, it calculates the next Fibonacci number (R10 = R11 + R12), then updates R12 to the old R11, and R11 to the new R10.`
  - `The final result is stored in RAX.`

Debugging Strategy:

* Initial Setup:

```
gdb ./your_executable
break asm_fibonachi
run
```

* Check Input N:

```
(gdb) p /d $rdi ; Check the value of N
```

* Test Base Cases First:
  - Call with N = 0. Set a breakpoint at .fib_is_zero_fib and verify RAX is 0.
  - Call with N = 1. Set a breakpoint at .fib_is_one_fib and verify RAX is 1.
* Step Through Iterative Loop (for N > 1):
  - Set a breakpoint at .forfib_user_fib.
  - At each iteration, examine the values of R8 (loop counter), R12 (F(k-2)), R11 (F(k-1)), and R10 (current Fibonacci number).

    ```
    (gdb) p /d $r8  ; loop counter
    (gdb) p /d $r12 ; F(k-2)
    (gdb) p /d $r11 ; F(k-1)
    (gdb) p /d $r10 ; current calculated fib
    ```

  - Trace the values of R11 and R12 to ensure they are being updated correctly to the previous two Fibonacci numbers.
  - Ensure the loop terminates at the correct iteration (CMP R8, R13).
* Verify Final Result:
  - Set a breakpoint at .fib_finish_fib or just before RET.
  - Check the final value of RAX to ensure it matches the expected Fibonacci number for the given 'N'.

3 Introduction

Integrating Assembly (ASM) code with C, especially when managed by a Makefile, is a common practice for tasks requiring high performance or direct hardware interaction. This section will elaborate on how the provided Makefile and main.c files facilitate this integration.