CPL ASSIGNMENT 4

# Name: SRIVIDHYA BHUKYA

# Roll: BT23CSE031

## Optimizing Assembly Code for Functionality and Performance

**Introduction**

In this document, we analyse and compare unoptimized and optimized assembly implementations of three common functions: **Addition**, **Factorial**, and **Inline Function (Square)**. By optimizing the assembly code, we aim to reduce instruction count, improve performance, and make the code more efficient in terms of execution time and space complexity. We will explore each function’s unoptimized and optimized versions, emphasizing the differences in structure, performance, and execution flow.

1. **Addition**

COMPILATION COMMANDS:

**Non-optimized : gcc -S -O0 main.c -o main\_no\_optimization.s**

**Optimisation: gcc -S -O2 main.c -o main\_optimization.s**

**Unoptimized Code (main\_no\_optimization.s)**

This program calculates the sum of two integers using the add function, which takes two integers as input, adds them, and returns the result. In the main function, the integers x = 3 and y = 7 are passed to add, and the result is stored in result. The program then prints the result using printf. In the unoptimized version, the add function has a separate stack frame, and parameters are passed via the stack, resulting in additional instructions for setup and teardown. The code looks as follows:

.file "main.c"

.text

# Function: \_addNumbers

.globl \_addNumbers

.def \_addNumbers; .scl 2; .type 32; .endef

\_addNumbers:

# Function prologue

LFB10:

.cfi\_startproc

pushl %ebp # Save the base pointer

movl %esp, %ebp # Set the new base pointer

# Load parameters and add them

movl 8(%ebp), %edx # Load the first parameter into %edx

movl 12(%ebp), %eax # Load the second parameter into %eax

addl %edx, %eax # Add the two parameters

# Function epilogue

popl %ebp # Restore the base pointer

ret # Return the result in %eax

.cfi\_endproc

LFE10:

# Data section for printf format string

.section .rdata,"dr"

LC0:

.ascii "The sum of %d and %d is %d\12\0" # Format string for printf

# Main function

.text

.globl \_main

.def \_main; .scl 2; .type 32; .endef

\_main:

# Function prologue

LFB11:

.cfi\_startproc

pushl %ebp # Save the base pointer

movl %esp, %ebp # Set the new base pointer

andl $-16, %esp # Align the stack

subl $32, %esp # Allocate space on the stack

# Prepare parameters for \_addNumbers

movl $5, 28(%esp) # First parameter (5)

movl $10, 24(%esp) # Second parameter (10)

# Call \_addNumbers

movl 24(%esp), %eax # Load second parameter into %eax

movl %eax, 4(%esp) # Push second parameter onto the stack

movl 28(%esp), %eax # Load first parameter into %eax

movl %eax, (%esp) # Push first parameter onto the stack

call \_addNumbers # Call the \_addNumbers function

# Store the result

movl %eax, 20(%esp) # Store the result in memory

# Prepare parameters for printf

movl 20(%esp), %eax # Load the result into %eax

movl %eax, 12(%esp) # Push the result onto the stack

movl 24(%esp), %eax # Load the second parameter into %eax

movl %eax, 8(%esp) # Push the second parameter onto the stack

movl 28(%esp), %eax # Load the first parameter into %eax

movl %eax, 4(%esp) # Push the first parameter onto the stack

movl $LC0, (%esp) # Push the format string onto the stack

call \_printf # Call printf to print the result

# Exit the program

movl $0, %eax # Set return value to 0

leave # Restore the stack and base pointer

ret # Return from main

.cfi\_endproc

LFE11:

.ident "GCC: (MinGW.org GCC-6.3.0-1) 6.3.0"

.def \_printf; .scl 2; .type 32; .endef

* **Explanation**: This assembly code represents a program that calculates the sum of two integers (5 and 10) using the \_addNumbers function and prints the result using printf. The \_addNumbers function performs the addition, while the main function prepares the parameters, calls \_addNumbers, and then uses printf to display the result. The unoptimized version includes additional stack operations and explicit memory management, making it less efficient but easier to follow.

**Optimized Code (main\_optimized.s)**

The optimized version of the addition function eliminates redundant operations.

.file "main.c"

.text

.p2align 4,,15

# Function: \_addNumbers

.globl \_addNumbers

.def \_addNumbers; .scl 2; .type 32; .endef

\_addNumbers:

# Function prologue

LFB12:

.cfi\_startproc

movl 8(%esp), %eax # Load the first parameter into %eax

addl 4(%esp), %eax # Add the second parameter to %eax

ret # Return the result in %eax

.cfi\_endproc

LFE12:

# Data section for printf format string

.section .rdata,"dr"

LC0:

.ascii "The sum of %d and %d is %d\12\0" # Format string for printf

# Main function

.section .text.startup,"x"

.p2align 4,,15

.globl \_main

.def \_main; .scl 2; .type 32; .endef

\_main:

# Function prologue

LFB13:

.cfi\_startproc

pushl %ebp # Save the base pointer

.cfi\_def\_cfa\_offset 8

.cfi\_offset 5, -8

movl %esp, %ebp # Set the new base pointer

.cfi\_def\_cfa\_register 5

andl $-16, %esp # Align the stack

subl $16, %esp # Allocate space on the stack

# Call the \_addNumbers function

movl $15, 12(%esp) # Push the first parameter (15) onto the stack

movl $10, 8(%esp) # Push the second parameter (10) onto the stack

movl $5, 4(%esp) # Push an unused value (5) onto the stack

movl $LC0, (%esp) # Push the format string onto the stack

call \_printf # Call printf to print the result

# Exit the program

xorl %eax, %eax # Set return value to 0

leave # Restore the stack and base pointer

.cfi\_restore 5

.cfi\_def\_cfa 4, 4

ret # Return from main

.cfi\_endproc

LFE13:

.ident "GCC: (MinGW.org GCC-6.3.0-1) 6.3.0"

.def \_printf; .scl 2; .type 32; .endef

* **Explanation**:The program calculates the sum of two integers (15 and 10) using the \_addNumbers function and prints the result using printf. The optimized assembly code minimizes stack operations and directly manipulates registers for efficiency.

**Performance Comparison**

| **Metric** | **Unoptimized Code** | **Optimized Code** |
| --- | --- | --- |
| **Instruction Count** | Higher due to redundant instructions like saving/restoring registers and stack setup | Lower, as redundant instructions are removed, and direct register manipulation is used |
| **Execution Time** | Slower due to additional stack operations and function overhead | Faster, as the compiler optimizes function calls and reduces unnecessary operations |
| **Memory Usage** | Higher , as more stack space is allocated and used for parameter passing | Lower, as parameters are directly passed via registers, reducing stack usage |

The unoptimized version has more stack operations and redundant instructions, while the optimized version minimizes overhead by using registers and reducing instruction count.

1. **Factorial**

COMPILATION COMMANDS:

**Non-optimized : gcc -S -O0 factorial.c -o factorial\_no\_optimization.s**

**Optimisation: gcc -S -O2 factorial.c -o factorial\_optimization.s**

**Unoptimized Code (factorial\_unoptimized.s)**

The unoptimized factorial function utilizes recursion with multiple stack operations (push/pop), which results in high overhead for each recursive call:

.file "factorial.c"

.text

.globl \_factorial

\_factorial:

# Function prologue

pushl %ebp # Save the base pointer

movl %esp, %ebp # Set the new base pointer

# Load input into %edx

movl 8(%ebp), %edx # Load the input parameter (n) into %edx

# Initialize result to 1 in %eax

movl $1, %eax # Set result = 1

# Check if input is 0

testl %edx, %edx # Check if n == 0

je .done # If n == 0, jump to .done

.loop:

# Multiply result by input

imull %edx, %eax # result \*= n

# Decrement input

subl $1, %edx # n--

# Continue loop if input is not 0

testl %edx, %edx # Check if n == 0

jne .loop # If n != 0, repeat the loop

.done:

# Function epilogue

popl %ebp # Restore the base pointer

ret # Return the result

.section .rdata,"dr"

LC0:

.ascii "Factorial of %d is %d\12\0"

.text

.globl \_main

\_main:

# Function prologue

pushl %ebp # Save the base pointer

movl %esp, %ebp # Set the new base pointer

# Prepare for factorial calculation

movl $5, %eax # Set num = 5

pushl %eax # Push num onto the stack

call \_factorial # Call the factorial function

addl $4, %esp # Clean up the stack

# Prepare for printf

movl %eax, 8(%esp) # Move result to the stack for printf

movl $5, 4(%esp) # Move num to the stack for printf

movl $LC0, (%esp) # Move format string to the stack for printf

call \_printf # Call printf

# Exit program

movl $0, %eax # Set return value to 0

leave # Restore the stack

ret # Return from main

.ident "GCC: (MinGW.org GCC-6.3.0-1) 6.3.0"

.def \_printf; .scl 2; .type 32; .endef

* **Explanation**: The function checks if the input is 0 (base case). If not, it recursively calls itself with the input decremented by 1, multiplying the result of each recursion.

**Optimized Code (factorial\_optimized.s)**

The optimized version eliminates recursion overhead, using a loop to calculate the factorial directly.

.file "factorial.c"

.text

.p2align 4,,15

.globl \_factorial

.def \_factorial; .scl 2; .type 32; .endef

\_factorial:

# Function prologue

pushl %ebp

movl %esp, %ebp

# Load input into %edx

movl 8(%ebp), %edx

# Initialize result to 1 in %eax

movl $1, %eax

# Check if input is 0

testl %edx, %edx

je .done

.loop:

# Multiply result by input

imull %edx, %eax

# Decrement input

subl $1, %edx

# Continue loop if input is not 0

testl %edx, %edx

jne .loop

.done:

# Function epilogue

popl %ebp

ret

.cfi\_endproc

LFE12:

.def \_\_\_main; .scl 2; .type 32; .endef

.section .rdata,"dr"

LC0:

.ascii "Factorial of %d is %d\12\0"

.section .text.startup,"x"

.p2align 4,,15

.globl \_main

.def \_main; .scl 2; .type 32; .endef

\_main:

LFB13:

.cfi\_startproc

pushl %ebp

.cfi\_def\_cfa\_offset 8

.cfi\_offset 5, -8

movl %esp, %ebp

.cfi\_def\_cfa\_register 5

andl $-16, %esp

subl $16, %esp

call \_\_\_main

movl $120, 8(%esp)

movl $5, 4(%esp)

movl $LC0, (%esp)

call \_printf

xorl %eax, %eax

leave

.cfi\_restore 5

.cfi\_def\_cfa 4, 4

ret

.cfi\_endproc

LFE13:

.ident "GCC: (MinGW.org GCC-6.3.0-1) 6.3.0"

.def \_printf; .scl 2; .type 32; .endef

* **Explanation**: The optimization replaces recursion with a simple loop, reducing stack usage and making the code more efficient.

**Performance Comparison**

| **Metric** | **Unoptimized Code** | **Optimized Code** |
| --- | --- | --- |
| **Instruction Count** | High (due to recursion) | Low (iterative loop) |
| **Execution Time** | Slower due to recursion | Faster, no recursion |
| **Memory Usage** | High due to stack calls | Low, iterative approach |

The optimized factorial function significantly reduces memory and time complexity by using a loop instead of recursion.

1. **Inline Function (Square)**

COMPILATION COMMANDS:

**Non-optimized : gcc -S -O0 square.c -o square\_no\_optimization.s**

**Optimisation: gcc -S -O2 square.c -o square\_optimization.s**

**Unoptimized Code (square\_no\_optimization.s)**

The unoptimized version of the square function is a simple function call with push and pop operations for saving registers.

.file "square.c"

# Data section for printf format string

.section .rdata,"dr"

LC0:

.ascii "The square of %d is %d\12\0" # Format string for printf

.text

# Main function

.globl \_main

.def \_main; .scl 2; .type 32; .endef

\_main:

LFB11:

.cfi\_startproc

pushl %ebp # Save the base pointer

movl %esp, %ebp # Set the new base pointer

andl $-16, %esp # Align the stack

subl $32, %esp # Allocate space on the stack

# Initialize the input value for square calculation

movl $4, 28(%esp) # Store the value 4 in memory

# Call the \_square function

movl 28(%esp), %eax # Load the input value into %eax

movl %eax, (%esp) # Push the input value onto the stack

call \_square # Call the \_square function

movl %eax, 24(%esp) # Store the result in memory

# Prepare parameters for printf

movl 24(%esp), %eax # Load the result into %eax

movl %eax, 8(%esp) # Push the result onto the stack

movl 28(%esp), %eax # Load the input value into %eax

movl %eax, 4(%esp) # Push the input value onto the stack

movl $LC0, (%esp) # Push the format string onto the stack

call \_printf # Call printf to print the result

# Exit the program

movl $0, %eax # Set return value to 0

leave # Restore the stack and base pointer

ret # Return from main

.cfi\_endproc

LFE11:

# Square function

.globl \_square

.def \_square; .scl 2; .type 32; .endef

\_square:

# Function prologue

pushl %ebp # Save the base pointer

movl %esp, %ebp # Set the new base pointer

# Calculate the square

movl 8(%ebp), %eax # Load the input value into %eax

imull %eax, %eax # Multiply the value by itself (square)

# Function epilogue

popl %ebp # Restore the base pointer

ret # Return the result in %eax

.cfi\_endproc

LFE12:

.ident "GCC: (MinGW.org GCC-6.3.0-1) 6.3.0"

.def \_printf; .scl 2; .type 32; .endef

* **Explanation**: The square function multiplies the input value by itself, but it is a separate function call, which adds overhead.

**Optimized Code (square\_optimized.s)**

The optimized version inlines the square function directly into the main function to avoid the overhead of function calls.

.file "square.c"

.def \_\_\_main; .scl 2; .type 32; .endef

.section .rdata,"dr"

LC0:

.ascii "The square of %d is %d\12\0"

.section .text.startup,"x"

.p2align 4,,15

.globl \_main

.def \_main; .scl 2; .type 32; .endef

\_main:

LFB13:

.cfi\_startproc

pushl %ebp

.cfi\_def\_cfa\_offset 8

.cfi\_offset 5, -8

movl %esp, %ebp

.cfi\_def\_cfa\_register 5

andl $-16, %esp

subl $16, %esp

call \_\_\_main

movl $16, 8(%esp)

movl $4, 4(%esp)

movl $LC0, (%esp)

call \_printf

xorl %eax, %eax

leave

.cfi\_restore 5

.cfi\_def\_cfa 4, 4

ret

.cfi\_endproc

LFE13:

.ident "GCC: (MinGW.org GCC-6.3.0-1) 6.3.0"

.def \_printf; .scl 2; .type 32; .endef

**Explanation**: The square calculation is done inline, removing the need for a function call and making the code more efficient.

**Performance Comparison**

| **Metric** | **Unoptimized Code** | **Optimized Code** |
| --- | --- | --- |
| **Instruction Count** | Higher (due to function call) | Lower (inlined code) |
| **Execution Time** | Slower (function call overhead) | Faster (direct calculation) |
| **Memory Usage** | Higher (function call stack) | Lower (no function call) |

Inlining the function reduces both instruction count and function call overhead, resulting in a more efficient implementation.

Finally,,

In this analysis, we compared the unoptimized and optimized versions of three functions: addition, Factorial, and Inline Function (Square). By reducing the number of instructions and eliminating unnecessary recursion and function calls, we significantly improved the performance and efficiency of the code. The key takeaways are:

* **Optimizations** lead to reduced instruction count and faster execution time.
* **Memory usage** is minimized by eliminating unnecessary function calls and stack operations.
* **Inlining functions** can dramatically improve performance when the overhead of function calls is significant.