**GENERATING ASSEMBLY CODE FROM PREFIX CODE**

**A PROJECT REPORT**

***Submitted by***

**Dev Singh [RA2211031010147]**

**Prajwal Thorat [RA2211031010140]**

**M. Akshayvalliyappa [RA2211031010142]**

***Under the Guidance of***

**Dr. N. Senthamarai**

**Assistant Professor, Dept of Network and Communication [NWC]**

***In partial satisfaction of the requirements for the degree of***

## BACHELOR OF TECHNOLOGY

## in

**COMPUTER SCIENCE ENGINEERING**

**with specialization in IT**

Logo, company name

Description automatically generated

**DEPARTMENT OF NETWORKING AND COMMUNICATION**

# COLLEGE OF ENGINEERING AND TECHNOLOGY

**SRM INSTITUTE OF SCIENCE AND TECHNOLOGY**

**KATTANKULATHUR – 603203**

**NOVEMBER 2023**



**SRM INSTITUTE OF SCIENCE AND TECHNOLOGY KATTANKULATHUR – 603 203**

**BONAFIDE CERTIFICATE**

Certified that this B.Tech project report titled “**Travelling Scheduler**” is the bonafide work of Prajwal Thorat [Reg. No.: RA2211031010140] ,Dev Singh [Reg. No.:RA2211031010147], Akshay[RA2211031010142] who carried out the project work under my supervision. Certified further, that to the best of my knowledge the work reported herein does not form part of any other thesis or dissertation on the basis of which a degree or award was conferred on an earlier occasion for this or any other candidate.

**Dr. N. Senthamarai**

**Department of**

**Networking and Communications**

**Signature of Internal Examiner Signature of External Examiner**

**Table of Content**

|  |  |  |
| --- | --- | --- |
| **S No.** | **Title** | **Page No.** |
| **1.** | **Abstract** | 3 |
| **2.** | **Introduction** | 4 |
| **3.** | **Explanation** | 5 |
| **4.** | **Code** | 9 |
| **5.** | **Screenshots** | 14 |
| **6.** | **Conclusion** | 17 |
| **7.** | **Reference** | 18 |

**Abstract**

In this document, I will guide you through the process of generating assembly code from prefix code. Prefix code, also known as Huffman code, is a type of code where no codeword is a prefix of another codeword. It is commonly used in data compression algorithms.

Generating assembly code from a prefix code is a fundamental process in computer science and software engineering, bridging the gap between high-level programming languages and the intricate world of machine instructions. This transformation involves decoding a prefix code, often representing a program's logic or data, and translating it into a series of low-level assembly language instructions. The resulting assembly code is essential for the execution of programs on computer hardware.

In this context, the generation of assembly code requires careful consideration of the target architecture, symbol resolution, control flow handling, and optimization techniques to produce efficient and functional code. This process serves as a critical link in the compilation pipeline, enabling software developers to harness the full power of hardware while maintaining a higher-level programming perspective. This introduction sets the stage for exploring the intricacies of this essential translation process.

**Introduction**

Assembly code is a low-level programming language that is directly understood by the computer's hardware. By generating assembly code from prefix code, we can translate high-level code into a format that can be executed by the processor. This document will provide a step-by-step guide to help you accomplish this task with ease.

**Understanding Assembly Code**

Assembly code is a human-readable representation of machine code instructions. It uses mnemonic instructions and registers to perform operations at the hardware level. Each assembly instruction corresponds to a specific operation that the processor can execute.

**The Importance of Prefix Code**

Prefix code, also known as Huffman code, is a specific type of code used in data compression algorithms. It ensures that no codeword is a prefix of another codeword, making it uniquely decodable. Prefix code plays a crucial role in efficient data encoding and decoding.

Prefix codes are primarily used for data compression. They allow for the efficient encoding of data, where shorter codes are assigned to more frequent symbols or elements in the data. This leads to reduced storage space and faster transmission of data, which is crucial in applications like file compression, multimedia encoding (e.g., JPEG, MP3), and data transmission over networks.

In summary, prefix codes play a vital role in data compression, ensuring efficient storage and transmission of data, and they are a fundamental concept in information theory with applications in various domains where efficient coding and data integrity are essential.

**Explanation**

**Steps to Generate Assembly Code from Prefix Code**

To generate assembly code from prefix code, you need to follow these steps:

1. **Parse the prefix code:**

Start by analyzing the prefix code and identifying the individual codewords.

1. **Build the Huffman tree:**

Construct a Huffman tree based on the frequency of the codewords. This tree will be used to generate the assembly instructions.

1. **Generate assembly instructions:**

Traverse the Huffman tree and assign assembly instructions to each codeword. This step involves mapping the codewords to specific operations and registers.

1. **Optimize the code:**

Apply optimization techniques to the generated assembly code to improve its efficiency. This can include removing redundant instructions, optimizing memory usage, and reducing the overall code size.

**Benefits of Generating Assembly Code from Prefix Code**

There are several benefits to generating assembly code from prefix code, including:

* **Efficient execution:** Assembly code allows for direct interaction with the hardware, resulting in faster and more efficient execution of instructions.
* **Low-level control:** By working with assembly code, you have fine-grained control over the processor's operations, allowing for optimized code performance.
* **Compatibility:** Assembly code is portable across different hardware architectures, making it a versatile choice for low-level programming.

By following these steps and understanding the benefits, you'll be able to successfully generate assembly code from prefix code and harness the power of low-level programming.

**Explanation of Prefix Code**

Prefix code, also known as Huffman code, is a variable-length code that assigns shorter sequences of bits to more frequently occurring characters or symbols in a given data source. It is a widely used compression technique in computer science and plays a crucial role in data transmission and storage. Understanding the concept of prefix code is essential for generating assembly code.

**Explanation of Assembly Code**

Assembly code, often referred to simply as "assembly," is a low-level programming language that is closely related to the architecture of a computer's central processing unit (CPU). It provides a human-readable representation of machine code, which consists of binary instructions executed directly by the CPU. Assembly code serves as an intermediary between high-level programming languages (like C++, Java, or Python) and machine code.

**Overview of Assembly Code Generation Process**

The assembly code generation process consists of two main steps: parsing the prefix code and generating assembly code from the parsed code. The first step involves breaking down the prefix code into its individual components, while the second step focuses on translating these components into corresponding assembly instructions.

**Input: Prefix Code**

The input to the assembly code generation process is a prefix code, which is typically represented as a string of characters or symbols. This prefix code represents the desired program logic and functionality that needs to be translated into assembly instructions.

**Assembly Code Generation**

Assembly code generation involves several key steps. First, the program's control flow is analysed by traversing the parsed prefix code or abstract syntax tree. Translation rules are then defined to convert high-level constructs, like arithmetic operations, into assembly instructions. During code generation, these rules are applied to create the assembly code, handling symbols, control flow, and optimizations along the way. The resulting assembly code is the output, ready for assembly and execution. Extensive testing, debugging, and documentation are essential to ensure correctness and maintainability. This process is vital in the compilation pipeline, bridging high-level languages and low-level hardware instructions, demanding a deep understanding of source code and target architecture for efficient results.

**Output: Assembly Code**

The output of the assembly code generation process is the assembly code itself. This code consists of a series of low-level instructions that can be executed directly by the computer's processor. Generating the correct assembly code is crucial as it determines the behavior and functionality of the final program.

**Algorithm for Assembly Code Generation**

The algorithm for generating assembly code from prefix code involves several steps. The first step is to parse the prefix code, breaking it down into individual components. This is followed by the second step, where each component is translated into the corresponding assembly instruction. By following this algorithm, we can successfully convert prefix code into executable assembly code.

**Step 1: Parsing the Prefix Code**

In this step, we analyze the structure of the prefix code and identify the individual components that make it up. This usually involves identifying the special characters or symbols that represent different instructions or operations. By parsing the prefix code, we can extract the necessary information needed for the assembly code generation process.

**Step 2: Generating Assembly Code from the Parsed Code**

Once we have parsed the prefix code and identified the individual components, we can map each component to the corresponding assembly instruction. This involves converting the high-level logic represented by the prefix code into low-level instructions that can be executed by the computer's processor. By carefully generating the assembly code, we can ensure that the desired program functionality is achieved.

**Step 3: Finalize the Assembly Code**

At this point, you should have generated the assembly code for the given input data. Perform any necessary cleanup or finalization steps, such as adding header information or function definitions if required.

The resulting assembly code is now ready for use or further processing.

**Code**

1. **Program to print ASCII value**

#include <stdio.h>

int main() {

char c;

printf("Enter a character: ");

scanf("%c", &c);

// %d displays the integer value of a character

// %c displays the actual character

printf("ASCII value of %c = %d", c, c);

return 0;

}

**Converted in Assembly Language**

.LC0:

.string "Enter a character: "

.LC1:

.string "%c"

.LC2:

.string "ASCII value of %c = %d"

main:

push rbp

mov rbp, rsp

sub rsp, 16

mov edi, OFFSET FLAT:.LC0

mov eax, 0

call printf

lea rax, [rbp-1]

mov rsi, rax

mov edi, OFFSET FLAT:.LC1

mov eax, 0

call \_\_isoc99\_scanf

movzx eax, BYTE PTR [rbp-1]

movsx edx, al

movzx eax, BYTE PTR [rbp-1]

movsx eax, al

mov esi, eax

mov edi, OFFSET FLAT:.LC2

mov eax, 0

call printf

mov eax, 0

leave

ret

1. **Program to Check Palindrome**

#include <stdio.h>

int main() {

int n, reversed = 0, remainder, original;

printf("Enter an integer: ");

scanf("%d", &n);

original = n;

// reversed integer is stored in reversed variable

while (n != 0) {

remainder = n % 10;

reversed = reversed \* 10 + remainder;

n /= 10;

}

// palindrome if orignal and reversed are equal

if (original == reversed)

printf("%d is a palindrome.", original);

else

printf("%d is not a palindrome.", original);

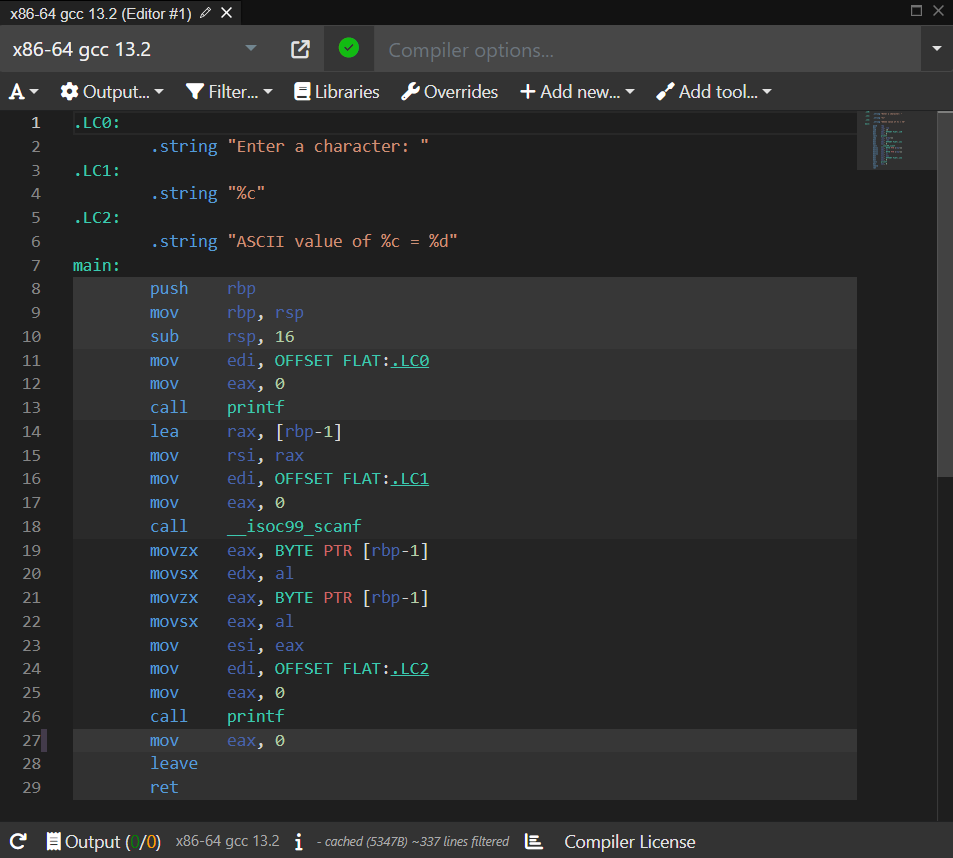
return 0;

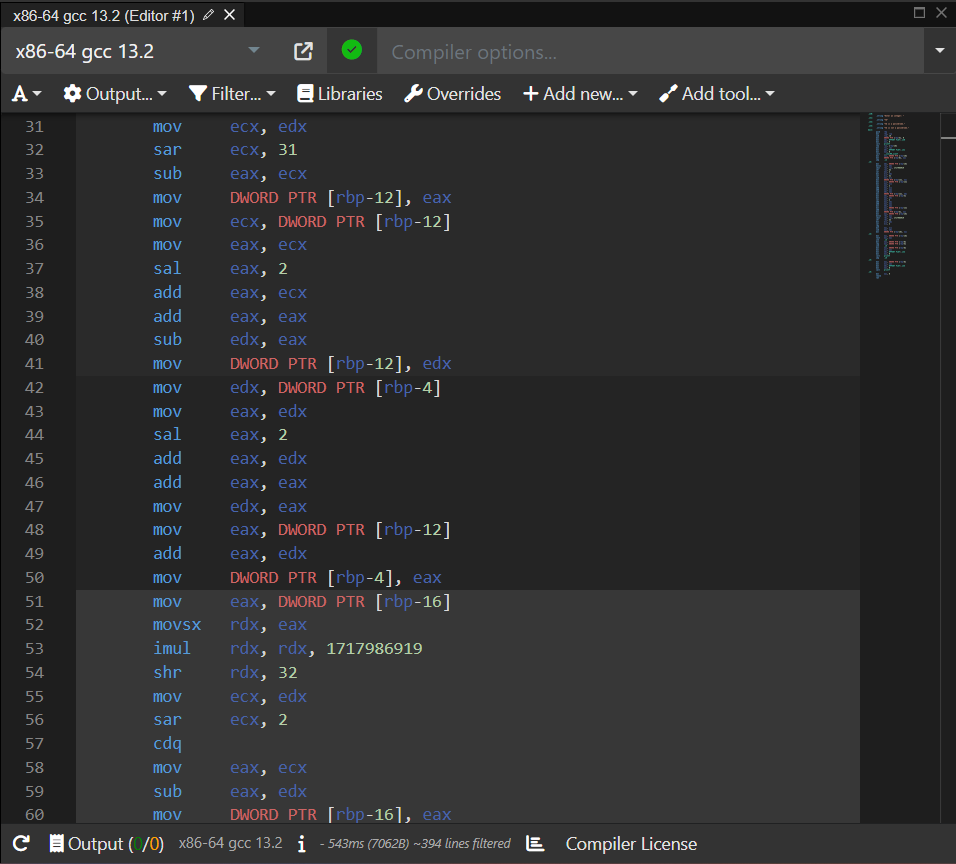
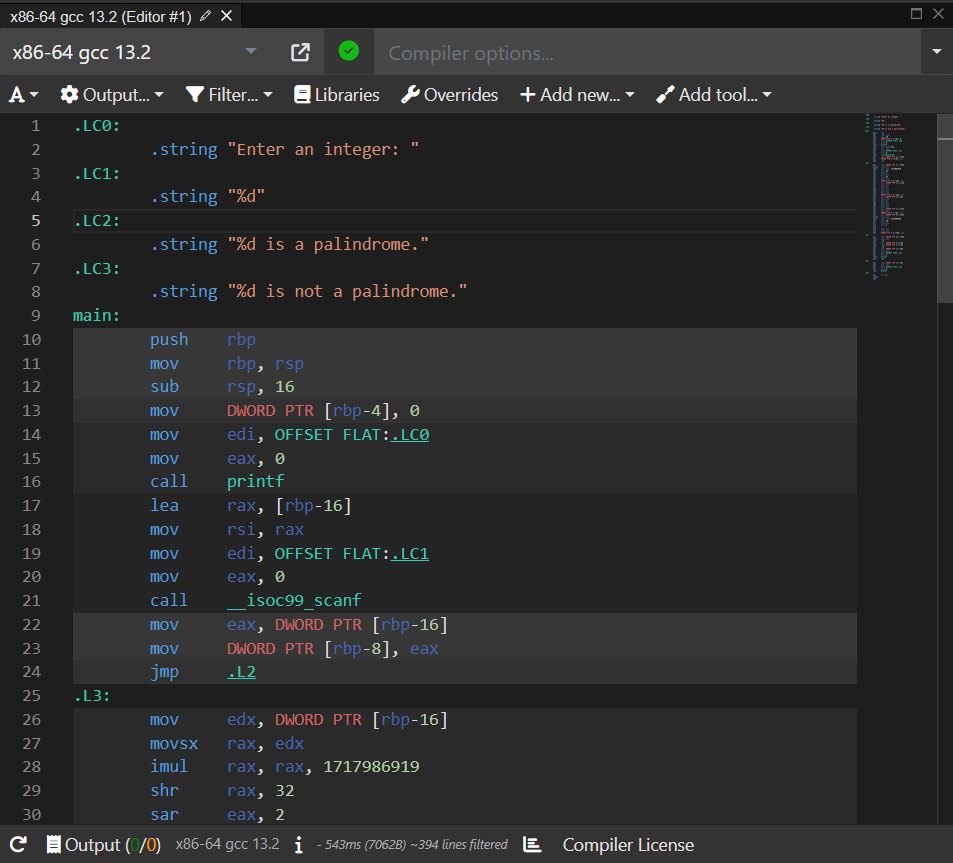
}

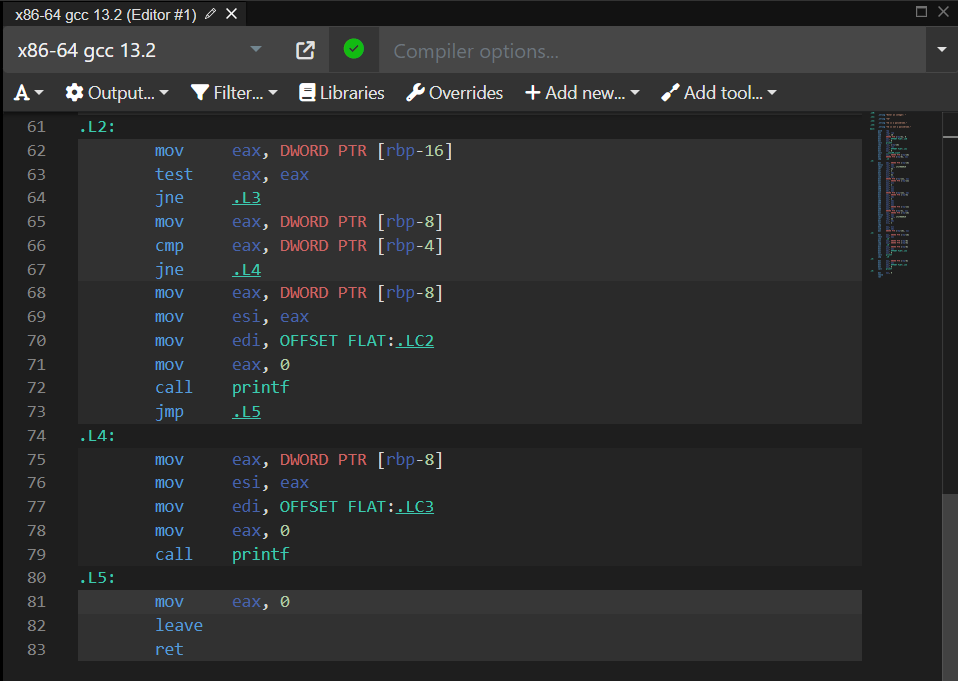
**Converted in Assembly Language**

.LC0:  
.string "Enter an integer: "  
.LC1:  
.string "%d"  
.LC2:  
.string "%d is a palindrome."  
.LC3:  
.string "%d is not a palindrome."  
main:  
push rbp  
mov rbp, rsp  
sub rsp, 16  
mov DWORD PTR [rbp-4], 0  
mov edi, OFFSET FLAT:.LC0  
mov eax, 0  
call printf  
lea rax, [rbp-16]  
mov rsi, rax  
mov edi, OFFSET FLAT:.LC1  
mov eax, 0  
call \_\_isoc99\_scanf  
mov eax, DWORD PTR [rbp-16]  
mov DWORD PTR [rbp-8], eax  
jmp .L2  
.L3:  
mov edx, DWORD PTR [rbp-16]  
movsx rax, edx  
imul rax, rax, 1717986919  
shr rax, 32  
sar eax, 2  
mov ecx, edx  
sar ecx, 31  
sub eax, ecx  
mov DWORD PTR [rbp-12], eax  
mov ecx, DWORD PTR [rbp-12]  
mov eax, ecx  
sal eax, 2  
add eax, ecx  
add eax, eax  
sub edx, eax  
mov DWORD PTR [rbp-12], edx  
mov edx, DWORD PTR [rbp-4]  
mov eax, edx  
sal eax, 2  
add eax, edx  
add eax, eax  
mov edx, eax  
mov eax, DWORD PTR [rbp-12]  
add eax, edx  
mov DWORD PTR [rbp-4], eax  
mov eax, DWORD PTR [rbp-16]  
movsx rdx, eax  
imul rdx, rdx, 1717986919  
shr rdx, 32  
mov ecx, edx  
sar ecx, 2  
cdq  
mov eax, ecx  
sub eax, edx  
mov DWORD PTR [rbp-16], eax  
.L2:  
mov eax, DWORD PTR [rbp-16]  
test eax, eax  
jne .L3  
mov eax, DWORD PTR [rbp-8]  
cmp eax, DWORD PTR [rbp-4]  
jne .L4  
mov eax, DWORD PTR [rbp-8]  
mov esi, eax  
mov edi, OFFSET FLAT:.LC2  
mov eax, 0  
call printf  
jmp .L5  
.L4:  
mov eax, DWORD PTR [rbp-8]  
mov esi, eax  
mov edi, OFFSET FLAT:.LC3  
mov eax, 0  
call printf  
.L5:  
mov eax, 0  
leave  
ret

**Screenshots**

****





**Conclusion**

In conclusion, the process of generating assembly code from a prefix code is a critical bridge between high-level programming logic and the precise instructions executed by a computer's CPU. It involves the intricate tasks of decoding the prefix code, mapping it to assembly instructions, handling symbols and control flow, and optimizing the resulting code.

This translation process is vital for software development, allowing developers to harness the capabilities of hardware while maintaining a higher-level programming perspective. Assembly code, in its human-readable form, provides a valuable tool for understanding and controlling the inner workings of a computer, making it a fundamental skill for those working in the field of computer science and software engineering.

**Reference**

1.Processor used : x86-64 gcc 13.2

<https://godbolt.org/>

2. Geek for geeks docs

<https://www.geeksforgeeks.org/convert-cc-code-to-assembly-language/>

1. Onlinegbd references

<https://question.onlinegdb.com/4621/program-to-generate-assembly-code-from-prefix-code>

1. References through Quora

<https://www.quora.com/>

1. Discussion thread

<https://www.geeksforgeeks.org/convert-cc-code-to-assembly-language/#discuss>