Cuarto Laboratorio GRUPO MACACO

Estudiantes:

Luis Mario Toscano Palomino.

Jhon Anderson Vargas Gómez.

Jeider Torres Martínez.

Samuel David Traslaviña Mateus.

Luis Alejandro Vecino Daza.

Profesor:

Carlos Jaime Barrios Hernández

Grupo:

A1

Arquitectura de computadores

Universidad Industrial de Santander

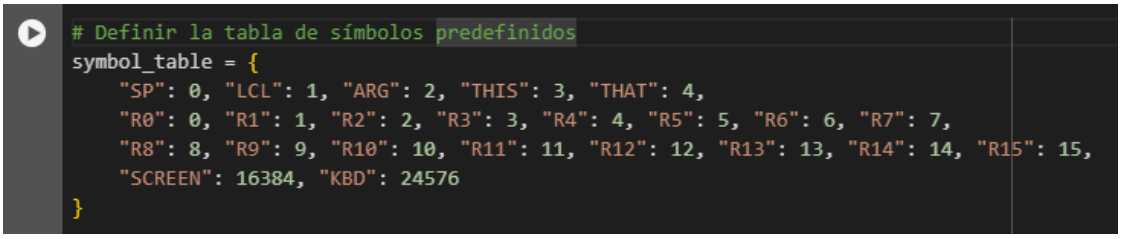
3 de octubre de 2024

Contenido

Tabla de símbolos predefinidos 3
Limpieza del código de entrada 3
Primer pase: Manejo de etiquetas 3
Traducción de instrucciones C a binario 4
Segundo pase: Traducción completa 5
Guardado del archivo de salida 6
Función principal para ensamblar 6
Carga y descarga de archivos en Colab 6

Tabla de símbolos predefinidos

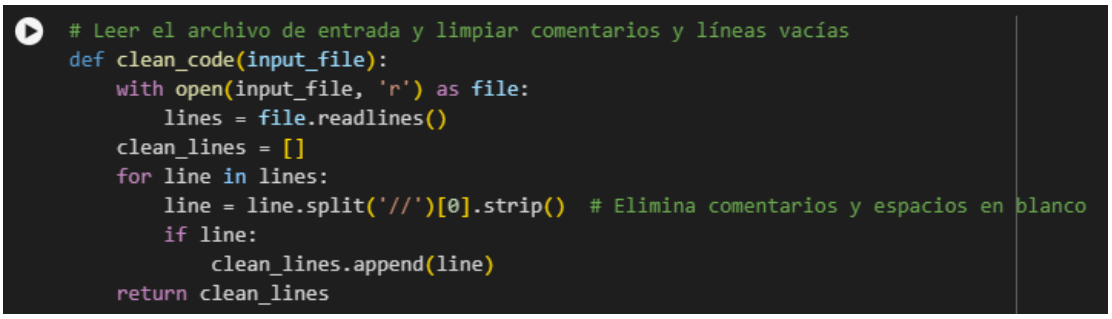
El código comienza definiendo una tabla de símbolos predefinidos llamada symbol_table. Esta tabla contiene los nombres de símbolos específicos del lenguaje ensamblador de Hack y sus correspondientes direcciones de memoria. Por ejemplo, SP apunta a la dirección 0, LCL a la dirección 1, ARG a la dirección 2, y así sucesivamente. También se incluyen los registros R0 a R15, que mapean directamente a las direcciones de memoria del 0 al 15. Por último, se definen dos direcciones importantes, SCREEN y KBD, que corresponden a la pantalla y el teclado de la máquina Hack, con valores de 16384 y 24576, respectivamente.



```
# Definir la tabla de símbolos predefinidos
symbol_table = {
    "SP": 0, "LCL": 1, "ARG": 2, "THIS": 3, "THAT": 4,
    "R0": 0, "R1": 1, "R2": 2, "R3": 3, "R4": 4, "R5": 5, "R6": 6, "R7": 7,
    "R8": 8, "R9": 9, "R10": 10, "R11": 11, "R12": 12, "R13": 13, "R14": 14, "R15": 15,
    "SCREEN": 16384, "KBD": 24576
}
```

Limpieza del código de entrada

La función clean_code se encarga de procesar el archivo ensamblador de entrada, eliminando comentarios y espacios en blanco. Para ello, abre el archivo, lee sus líneas y luego procesa cada una eliminando los comentarios que comienzan con //. También elimina cualquier espacio en blanco, asegurándose de que solo se conserven las líneas que contienen instrucciones válidas. El resultado es una lista de líneas "limpias" que serán utilizadas en el proceso de ensamblado.



```
# Leer el archivo de entrada y limpiar comentarios y líneas vacías
def clean_code(input_file):
    with open(input_file, 'r') as file:
        lines = file.readlines()
    clean_lines = []
    for line in lines:
        line = line.split('//')[0].strip()  # Elimina comentarios y espacios en blanco
        if line:
            clean_lines.append(line)
    return clean_lines
```

Primer pase: Manejo de etiquetas

En el primer pase, la función first_pass maneja las etiquetas, que en Hack son líneas que comienzan con un paréntesis, como (LOOP). Estas etiquetas son símbolos que representan direcciones de memoria a las que el código puede saltar durante la ejecución. La función recorre las líneas limpiadas y, cuando encuentra una etiqueta, la añade a la tabla de símbolos con el número de línea correspondiente. Cabe destacar que las etiquetas no se cuentan como instrucciones que ocupan espacio en la memoria, por lo que no incrementan el contador de línea.

```python
# Primer pase: Manejo de etiquetas
def first_pass(clean_lines):
    line_number = 0
    for line in clean_lines:
        if line.startswith("("):  # Esto es una etiqueta
            symbol = line[1:-1]  # Eliminar paréntesis
            symbol_table[symbol] = line_number
        else:
            line_number += 1
```

Traducción de instrucciones C a binario

La función translate_c_instruction se utiliza para traducir las instrucciones de tipo C (aquellas que realizan operaciones aritméticas y lógicas, o saltos condicionales) a su equivalente en código binario. Esta función utiliza tres diccionarios que contienen las representaciones binarias de las diferentes partes de una instrucción C: comp, dest y jump. comp representa la operación que se está realizando (por ejemplo, D+1), dest indica el destino donde se almacenará el resultado (por ejemplo, D, M), y jump define las condiciones bajo las cuales debe realizarse un salto (por ejemplo, JGT, JMP). La salida de esta función es una cadena de 16 bits que comienza con 111, seguido por los bits que representan comp, dest y jump.}

```python
# Traducción de instrucciones C a binario
def translate_c_instruction(dest, comp, jump):
    # Mapeo de las partes comp, dest y jump
    comp_dict = {
        "0": "0101010", "1": "0111111", "-1": "0111010",
        "D": "0001100", "A": "0110000", "M": "1110000",
        "!D": "0001101", "!A": "0110001", "!M": "1110001",
        "-D": "0001111", "-A": "0110011", "-M": "1110011",
        "D+1": "0011111", "A+1": "0110111", "M+1": "1110111",
        "D-1": "0001110", "A-1": "0110010", "M-1": "1110010",
        "D+A": "0000010", "D+M": "1000010", "D-A": "0010011",
        "D-M": "1010011", "A-D": "0000111", "M-D": "1000111",
        "D&A": "0000000", "D&M": "1000000", "D|A": "0010101",
        "D|M": "1010101"
    }
    dest_dict = {
        "": "000", "M": "001", "D": "010", "MD": "011", "A": "100", "AM": "101", "AD": "110", "AMD": "111"
    }
    jump_dict = {
        "": "000", "JGT": "001", "JEQ": "010", "JGE": "011",
        "JLT": "100", "JNE": "101", "JLE": "110", "JMP": "111"
    }
    comp_bits = comp_dict.get(comp, "0000000")
    dest_bits = dest_dict.get(dest, "000")
    jump_bits = jump_dict.get(jump, "000")

    return f"111{comp_bits}{dest_bits}{jump_bits}"
```

Segundo pase: Traducción completa

En el segundo pase, la función second_pass se encarga de traducir todas las instrucciones ensambladas a código binario. Para las instrucciones A, que comienzan con @, la función traduce el valor a una dirección de memoria en binario. Si el valor es numérico, simplemente lo convierte a binario. Si es un símbolo, se busca en la tabla de símbolos. Si no se encuentra, se asigna una nueva dirección de variable a partir de la posición 16. Las instrucciones C se traducen utilizando la función translate_c_instruction. El resultado es una lista de líneas de código binario que representan el programa en su forma final.

```
# Segundo pase: Traducción completa
def second_pass(clean_lines):
    current_variable_address = 16
    binary_lines = []

    for line in clean_lines:
        if line.startswith("@"):  # Instrucción A
            symbol = line[1:]
            if symbol.isdigit():
                address = int(symbol)
            elif symbol in symbol_table:
                address = symbol_table[symbol]
            else:
                symbol_table[symbol] = current_variable_address
                address = current_variable_address
                current_variable_address += 1
            binary_lines.append(f'{address:016b}')

        elif "=" in line or ";" in line:  # Instrucción C
            if "=" in line:
                dest, rest = line.split("=")
            else:
                dest = ""
                rest = line

            if ";" in rest:
                comp, jump = rest.split(";")
            else:
                comp = rest
                jump = ""

            binary_line = translate_c_instruction(dest, comp, jump)
            binary_lines.append(binary_line)

    return binary_lines
```

Guardado del archivo de salida

La función write_to_file toma las líneas de código binario generadas durante el segundo pase y las guarda en un archivo de salida con extensión .hack. Este archivo es el que puede ser cargado en la máquina Hack para ejecutar el programa. Cada línea de código binario se escribe en una nueva línea del archivo.

```
# Guardar el archivo de salida
def write_to_file(output_file, binary_lines):
    with open(output_file, 'w') as file:
        for line in binary_lines:
            file.write(line + '\n')
```

Función principal para ensamblar

La función assembler es la función principal que coordina el proceso de ensamblado. Esta función llama a clean_code para limpiar el código de entrada, luego realiza el primer pase con first_pass para manejar las etiquetas, seguido del segundo pase con second_pass para traducir las instrucciones a binario. Finalmente, llama a write_to_file para escribir el archivo de salida en formato .hack.

```
# Función principal para ensamblar el archivo
def assembler(input_file, output_file):
    clean_lines = clean_code(input_file)
    first_pass(clean_lines)
    binary_lines = second_pass(clean_lines)
    write_to_file(output_file, binary_lines)
```

Carga y descarga de archivos en Colab

En la parte final del código, se incluyen instrucciones específicas para el entorno de Google Colab, donde se permite al usuario subir archivos .asm y descargar el archivo .hack generado automáticamente. Esto es útil para realizar las pruebas en un entorno en línea, sin necesidad de trabajar localmente.

```python
from google.colab import files

# Subir archivo .asm desde la computadora
uploaded = files.upload()

# Nombre del archivo ensamblador subido (puede ser "Add.asm", "Max.asm", etc.)
input_file = "Add.asm"
output_file = "output.hack"

# Llamar a la función del ensamblador (el código del ensamblador debe estar definido previamente)
assembler(input_file, output_file)

# Descargar el archivo .hack generado
files.download(output_file)

print(f"Archivo {output_file} generado y descargado automáticamente en la carpeta de descargas.")
```