**Національний технічний університет України**

**“Київський політехнічний інститут ім. Ігоря Сікорського”**

**Факультет прикладної математики**

**Кафедра системного програмування і спеціалізованих комп’ютерних систем**

**ЛАБОРАТОРНА РОБОТА №2**

***з дисципліни***

***“Архітектура комп’ютерів ”***

**ТЕМА: “Перетворення віртуальних адрес”**

**Група: КВ-13**

**Виконав: Луценко Б. А.**

**Оцінка:**

**Київ – 2024**

**Мета роботи**

• ознайомитись із елементами рівня архітектури системи команд;

• ознайомитись з елементами рівня архітектури операційної системи на прикладі функції реалізації і підтримки віртуальної пам’яті; навчитись перетворювати віртуальні адреси у фізичні.

**Постановка задачі**

Завдання лабораторної роботи наступне: реалізувати програму мовою C або C++, що

виконує зчитування послідовності команд (програми) з файлу і заміняє віртуальні адреси на

фізичні в командах, що визначаються варіантом. Тип організації пам’яті також визначається

варіантом. Заміна адреси відбувається у випадку, якщо сторінка та/або сегмент знаходиться в

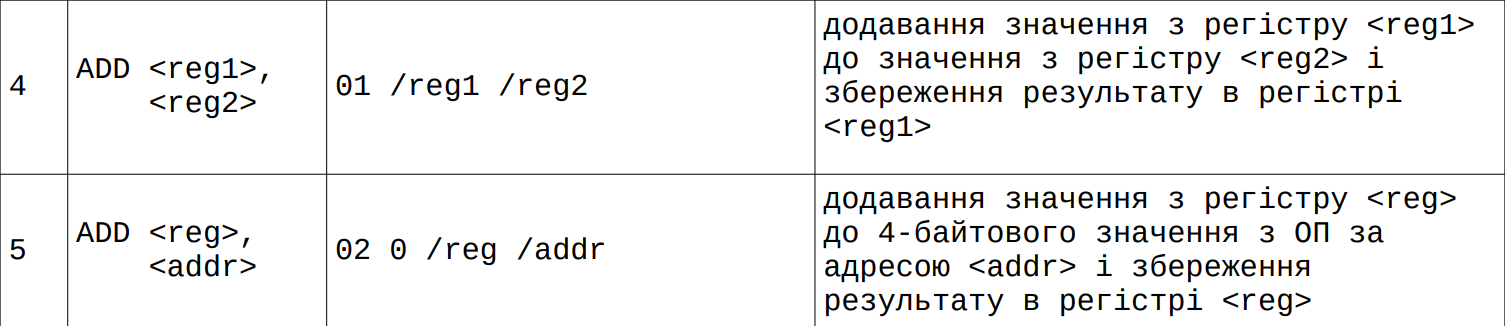
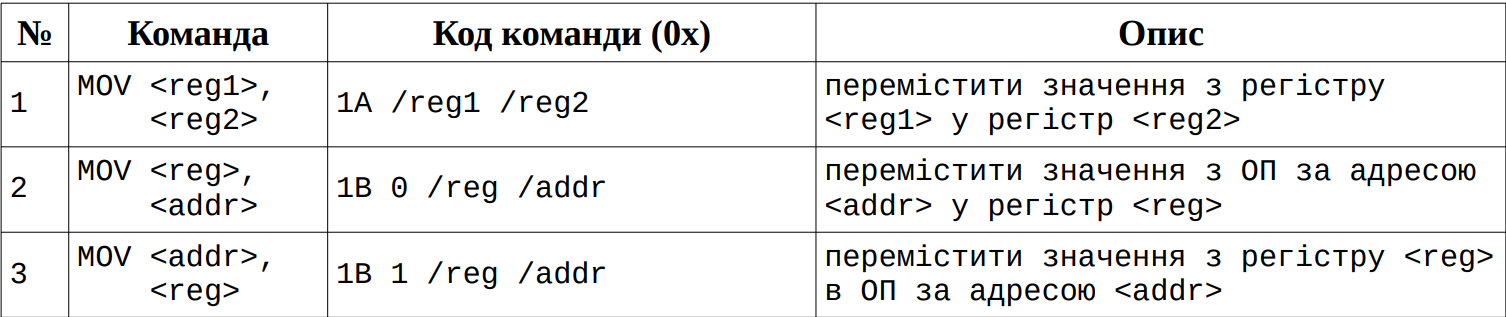
оперативній пам’яті (ОП). Якщо потрібна віртуальна сторінка та/або сегмент відсутні в ОП, тоді

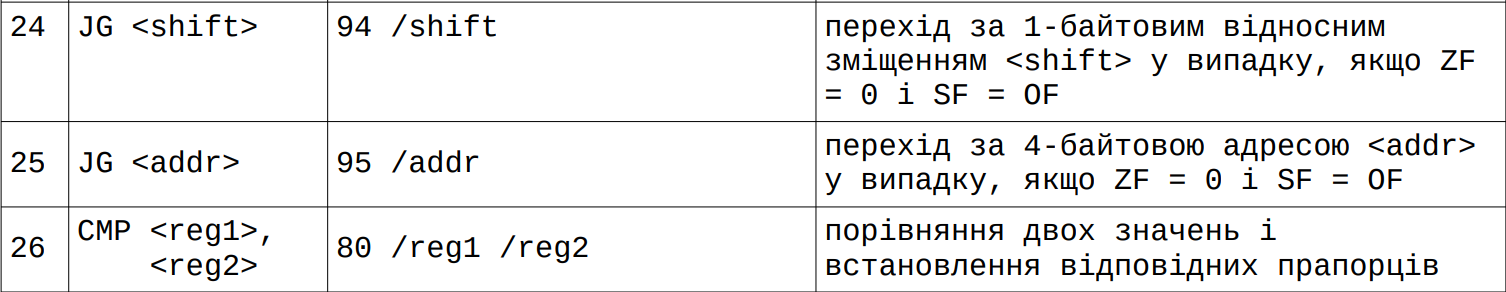
має бути виведене повідомлення про помилку відсутності сторінки/сегменту, й аналіз команд

має бути продовжено. Таблиця сторінок/сегментів задається у файлі формату CSV.

**Завдання за варіантом**









**Код програми**

**address.py**

import src.config as cfg

from src.exceptions import PageNotExistsInROMError, PageNotExistsInRAMError, SegmentNotExistsInDescriptorTableError, InvalidMemoryAddressError

import csv

# segments descriptors table contains the following fields:

# 1. segment number

# 2. segment pages table path

# 3. pages count

segments\_descriptors\_table = []

# segment pages table contains the following fields:

# 1. page number

# 2. existence bit

# 3. frame number

segment\_pages\_table = []

def load\_segments\_table():

    with open(cfg.SEGMENTS\_TABLE\_FILE, "r") as f:

        reader = csv.reader(f)

        for row in reader:

            segments\_descriptors\_table.append(row)

def load\_pages\_table(segment\_number):

    try:

        with open(cfg.DATA\_DIR + segments\_descriptors\_table[segment\_number][1] + ".csv", "r") as f:

            reader = csv.reader(f)

            for row in reader:

                segment\_pages\_table.append(row)

    except IndexError:

        raise SegmentNotExistsInDescriptorTableError(segment\_number)

def form\_phys\_addr(virtual\_addr: str) -> str:

    """

    Translates a virtual address to a physical address.

    The virtual address is a 32-bit value, divided into three parts:

    - The first 11 bits (0-10) represent the page offset.

    - The next 10 bits (11-20) represent the page number.

    - The final 11 bits (21-31) represent the segment number.

    The translation process is as follows:

    1. Retrieve the segment descriptor using the segment number.

    2. Retrieve the page table from the segment descriptor.

    3. Retrieve the frame number from the page table.

    4. Concatenate the frame number with the page offset to form the physical address.

    5. Return the physical address.

    Args:

        virtual\_address (int): The virtual address to be translated.

    Returns:

        int: The translated physical address.

    """

    # check if the address is valid

    if len(virtual\_addr) != 8:

        raise InvalidMemoryAddressError(virtual\_addr)

    try:

        binary = f'{int(virtual\_addr, 16):032b}'

    except ValueError:

        raise InvalidMemoryAddressError(virtual\_addr)

    page\_offset = int(binary[:11], 2)

    page\_number = int(binary[11:21], 2)

    segment\_number = int(binary[21:], 2)

    load\_segments\_table()

    load\_pages\_table(segment\_number)

    # check if the page is loaded

    try:

        if segment\_pages\_table[page\_number][1] == '0':

            raise PageNotExistsInRAMError(virtual\_addr)

    except IndexError:

        raise PageNotExistsInROMError(virtual\_addr)

    frame\_number = int(segment\_pages\_table[page\_number][2], 2)

    return f'{int(f"{frame\_number:021b}{page\_offset:011b}", 2):04X}'

**analyzer.py**

import src.config as cfg

from src.address import form\_phys\_addr

from src.exceptions import InvalidMemoryAddressError, PageNotExistsInRAMError, SegmentNotExistsInDescriptorTableError, PageNotExistsInROMError

def form\_addr(tokens: list[str]) -> str:

    virtual\_addr = ''

    for token in tokens:

        virtual\_addr += token

    return f'[{form\_phys\_addr(virtual\_addr)}]'

def clean\_data(data: str) -> str:

    return "".join(data.split())

def get\_tokens(data: str) -> list[str]:

    return [data[i:i+2] for i in range(0, len(data), 2)]

def print\_row(tokens: list[str], mnemonic: str, operands: int, error: str = None, file=None):

    print(' '.join(tokens), file=file)

    if error is not None:

        print(f'{error}', file=file)

    print(f'{mnemonic} {", ".join(operands)}', file=file); print(file=file)

def analyze(input\_file, output\_dir):

    print("Analyzing file:", input\_file)

    print("Output directory:", output\_dir)

    CLEANED\_FILE = output\_dir + "program\_cleaned.txt"

    ANALYSIS\_FILE = output\_dir + "analysis.txt"

    # read the input file and clean the data from whitespaces

    with open(input\_file, "r") as f:

        data = f.read()

        data = clean\_data(data)

    # write cleaned data to program\_cleaned.txt

    output\_file = CLEANED\_FILE

    with open(output\_file, "w") as f:

        f.write(data)

    # read the cleaned data assuming 2 characters as a single token - byte

    with open(output\_file, "r") as f:

        data = f.read()

        tokens = get\_tokens(data)

    with open(ANALYSIS\_FILE, "w") as f:

        # analyze the tokens

        i = 0

        command = []

        current\_token = ''

        arguments = []

        reverse\_flag = False

        while i < len(tokens):

            error = None

            current\_token = tokens[i]

            if current\_token in cfg.OPCODES:

                opcode = current\_token

                i += 1

                # i: index of the first operand

                command = [opcode]

                if opcode in cfg.REVERSE\_OPCODE:

                    reverse\_flag = tokens[i][0] == '0'

                operands = cfg.OPCODES[opcode]['operands']

                if len(operands) == 1:

                    operand\_size = cfg.OPERAND\_SIZES[operands[0]]

                    if operands[0] == 'SHIFT':

                        arguments = [f'{int(tokens[i], 16)}']

                        command.append(tokens[i])

                    elif operands[0] == 'ADDR':

                        try:

                            arguments = [form\_addr(tokens[i:i+operand\_size])]

                        except Exception as e:

                            error = str(e)

                            arguments = [f'[0x{"".join(tokens[i:i+operand\_size])}]']

                        command.extend(tokens[i:i+operand\_size])

                    i += operand\_size

                elif len(operands) == 2:

                    operands\_sizes = [cfg.OPERAND\_SIZES[operand] for operand in operands]

                    if operands[0] == 'REG' and operands[1] == 'REG':

                        arguments = [f'R{int(tokens[i][1], 16)}',

                                     f'R{int(tokens[i][0], 16)}']

                        command.append(tokens[i])

                        i += 1  # REG REG

                    elif operands[0] == 'REG' and operands[1] == 'ADDR':

                        command.append(tokens[i])

                        try:

                            arguments = [form\_addr(tokens[i+1:i+operands\_sizes[1]+1]),

                                         f'R{int(tokens[i][1], 16)}']

                        except Exception as e:

                            error = str(e)

                            arguments = [f'[0x{"".join(tokens[i+1:i+operands\_sizes[1]+1])}]',

                                         f'R{int(tokens[i][1], 16)}']

                        command.extend(tokens[i+1:i+operands\_sizes[1]+1])

                        i += 1                  # REG

                        i += operands\_sizes[1]  # ADDR

                    elif operands[0] == 'REG' and operands[1] == 'LIT16':

                        arguments = [f'R{int(tokens[i][1], 16)}',

                                     f'{int(tokens[i+1], 16)}']

                        command.append(tokens[i])

                        command.append(tokens[i+1])

                        i += 1 # REG

                        i += 2 # LIT16

                    if reverse\_flag:

                        arguments = arguments[::-1]

                        reverse\_flag = False

                print\_row(command,

                          cfg.OPCODES[opcode]['mnemonic'],

                          arguments,

                          error=error,

                          file=f)

    print("Analysis complete")

**config.py**

INPUT\_DIR = "input/"

OUTPUT\_DIR = "output/"

DATA\_DIR = "data/"

INPUT\_FILE = INPUT\_DIR + "program.txt"

OUTPUT\_FILE = OUTPUT\_DIR + "result.txt"

SEGMENTS\_TABLE\_FILE = DATA\_DIR + "segments.csv"

OPERAND\_SIZES = {

    'REG': 1,

    'ADDR': 4,

    'SHIFT': 1,

    'LIT16': 4

}

OPCODES = {

    '1A': {'mnemonic': 'MOV', 'operands': ['REG', 'REG']},

    '1B': {'mnemonic': 'MOV', 'operands': ['REG', 'ADDR']},

    '01': {'mnemonic': 'ADD', 'operands': ['REG', 'REG']},

    '02': {'mnemonic': 'ADD', 'operands': ['REG', 'ADDR']},

    '94': {'mnemonic': 'JG',  'operands': ['SHIFT']},

    '95': {'mnemonic': 'JG',  'operands': ['ADDR']},

    '80': {'mnemonic': 'CMP', 'operands': ['REG', 'REG']},

    '1C': {'mnemonic': 'MOV', 'operands': ['REG', 'LIT16']},

}

REVERSE\_OPCODE = [

    '1B',

    '02',

    '1C'

]

PAGE\_SIZE = 2\*2\*\*10

DESCRIPTOR\_TABLE\_SIZE = 2048

**exception.py**

class CustomError(Exception):

    """Base class for other exceptions"""

    pass

# MEMORY ERRORS

class PageNotExistsInRAMError(CustomError):

    """Raised when the page does not exist in RAM"""

    def \_\_init\_\_(self, virtual\_address: str , message="Page does not exist in RAM"):

        self.message = message + f" for address {virtual\_address}"

        super().\_\_init\_\_(self.message)

class PageNotExistsInROMError(CustomError):

    """Raised when the page does not exist in memory"""

    def \_\_init\_\_(self, page\_number: int, message="Page does not exist in memory"):

        self.message = message + f" for page number {page\_number}"

        super().\_\_init\_\_(self.message)

class SegmentNotExistsInDescriptorTableError(CustomError):

    """Raised when the segment does not exist in the descriptor table"""

    def \_\_init\_\_(self, segment\_number: int, message="Segment does not exist in the descriptor table"):

        self.message = message + f" for segment number {segment\_number}"

        super().\_\_init\_\_(self.message)

class InvalidMemoryAddressError(CustomError):

    """Raised when the memory address is invalid"""

    def \_\_init\_\_(self, virtual\_address: str, message="Invalid memory address"):

        self.message = message + f" for address {virtual\_address}"

        super().\_\_init\_\_(self.message)

**Тестування програми**

**program.txt**

1A 01

1A 9F

1A FF

1B 00 00000001

1B 05 1000FF00

1B 05 00070101

1B 10 00C000A1

1B 15 1000FF00

1B 15 00070101

01 01

01 9F

01 FF

02 00 00C000A1

02 05 1000FF00

02 05 00070101

94 03

94 09

94 0D

94 0F

94 10

94 1F

94 FF

95 00C000A1

95 1000FF00

95 00070101

80 01

80 9F

80 FF

1C 10 1234

1C 18 1234

1C 1D 1234

1C 1F 1234

1C 1F 0001

1C 1F 0100

**analysis.txt**

1A 01

MOV R1, R0

1A 9F

MOV R15, R9

1A FF

MOV R15, R15

1B 00 00 00 00 01

MOV R0, [DD565000]

1B 05 10 00 FF 00

Segment does not exist in the descriptor table for segment number 1792

MOV R5, [0x1000FF00]

1B 05 00 07 01 01

Segment does not exist in the descriptor table for segment number 257

MOV R5, [0x00070101]

1B 10 00 C0 00 A1

Segment does not exist in the descriptor table for segment number 161

MOV [0x00C000A1], R0

1B 15 10 00 FF 00

Segment does not exist in the descriptor table for segment number 1792

MOV [0x1000FF00], R5

1B 15 00 07 01 01

Segment does not exist in the descriptor table for segment number 257

MOV [0x00070101], R5

01 01

ADD R1, R0

01 9F

ADD R15, R9

01 FF

ADD R15, R15

02 00 00 C0 00 A1

Segment does not exist in the descriptor table for segment number 161

ADD R0, [0x00C000A1]

02 05 10 00 FF 00

Segment does not exist in the descriptor table for segment number 1792

ADD R5, [0x1000FF00]

02 05 00 07 01 01

Segment does not exist in the descriptor table for segment number 257

ADD R5, [0x00070101]

94 03

JG 3

94 09

JG 9

94 0D

JG 13

94 0F

JG 15

94 10

JG 16

94 1F

JG 31

94 FF

JG 255

95 00 C0 00 A1

Segment does not exist in the descriptor table for segment number 161

JG [0x00C000A1]

95 10 00 FF 00

Segment does not exist in the descriptor table for segment number 1792

JG [0x1000FF00]

95 00 07 01 01

Segment does not exist in the descriptor table for segment number 257

JG [0x00070101]

80 01

CMP R1, R0

80 9F

CMP R15, R9

80 FF

CMP R15, R15

1C 10 12

MOV R0, 18

1C 18 12

MOV R8, 18

1C 1D 12

MOV R13, 18

1C 1F 12

MOV R15, 18

1C 1F 00

MOV R15, 0

1C 1F 01

MOV R15, 1