Міністерство освіти і науки України
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет прикладної математики
Кафедра системного програмування
і спеціалізованих комп’ютерних системи

Лабораторна робота №2

з дисципліни
«Архітектура для програмістів»

Виконав:
студент групи КВ-11
Терентьєв Іван Дмитрович

Перевірив:
Молчанов О. А.

Київ – 2024

Загальне завдання

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.

Програма має містити наступні компоненти:

1.Модуаль зчитування таблиці сторінок/сегментів з файлу CSV і створення внутрішнього представлення відповідної таблиці (або таблиць)

2.Модуль з реалізацією функцій зчитування, аналізу і зміни команд з текстового файлу, що виконує заміну віртуальних адрес в зчитаних командах на фізичні

3.Модуль тестування, що містить тестові утиліти і тести реалізованої програми.

Завдання за варіантом 23

Тип організації пам'яті: сегментний.

РДТ: 2048(2^11)

Список команд: 1,2,4,5,21,23,25,26,29

	Команда	**Код команди (0x)**
	MOV<reg1>, <reg2>	1A /reg1 /reg2
	MOV<reg>, <addr>	1B 0 /reg /addr
	ADD<reg1>, <reg2>	01 /reg1 /reg2
	ADD<reg>, <addr>	02 0 /reg /addr
	JMP<addr>	91 /addr
	JL<addr>	93 /addr
	JG<addr>	95 /addr
	CMP<reg1>, <reg2>	80 /reg1 /reg2
	MOV<reg>, <lit32>	1C 2 /reg /lit32

<reg> - регістр загального призначення

<addr> - віртуальна адреса

<lit32> - константа 32-біт

/reg - номер регістру, 4 біти

/addr - адреса, 32 біти

/lit32 - беззнакове число розміром 32 біти

Лістинг програми мовою C

```
==> cli.c <==
#include "cli.h"
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

#define chck(str)                   \
    if (i + 1 < argc)               \
    {                               \
        my_files->str = argv[i + 1]; \
        i++;                        \
    }                               \
    else                            \
    {                               \
        return false;               \
    }

void show_help()
{
    printf("\n===Help===\n");
    printf("Required:\n");
    printf("-i or --input  <filename>     - input file\n");
    printf("-o or --output <filename>     - output file\n");
    printf("-t or --table  <filename>     - table file\n");
    printf("Optional:\n");
    printf("-q or --quiet                 - quiet mode\n");
    printf("-v or --verify <filename>     - verification file\n");
    printf("-h or --help                  - this help\n");
}

bool proc_cli(int argc, char *argv[], Files *my_files)
{
    for (int i = 1; i < argc; i++)
    {
        if (strcmp(argv[i], "-i") == 0){
            chck(_input);
        }
        if (strcmp(argv[i], "-o") == 0){
            chck(_output);
        }
        if (strcmp(argv[i], "-t") == 0){
            chck(_table);
        }
        if (strcmp(argv[i], "-v") == 0){
            chck(_verify);
        }
```

```
    if (strcmp(argv[i], "-q") == 0){
        my_files->verbose = false;
    }
    if (strcmp(argv[i], "-h") == 0)
    {
        show_help();
        if (argc == 2)
            return false;
    }

    if (strcmp(argv[i], "--input") == 0){
        chck(_input);
    }
    if (strcmp(argv[i], "--output") == 0){
        chck(_output);
    }
    if (strcmp(argv[i], "--table") == 0){
        chck(_table);
    }
    if (strcmp(argv[i], "--verify") == 0){
        chck(_verify);
    }
    if (strcmp(argv[i], "--quiet") == 0){
        my_files->verbose = false;
    }
    if (strcmp(argv[i], "--help") == 0)
    {
        show_help();
        if (argc == 2)
            return false;
    }
}
bool verified = true;
if (my_files->_input == NULL)
{
    printf("No input file\n");
    verified = false;
}
if (my_files->_output == NULL)
{
    printf("No output file\n");
    verified = false;
}
if (my_files->_table == NULL)
{
    printf("No table file\n");
    verified = false;
}
if (my_files->_verify)
```

```
        printf("Output will be verified\n");

    return verified;
}
==> converter-commands.c <==
#include "converter.h"
#include <string.h>
#include <stdlib.h>

size_t MOVREGREG(OUT *result, uint8_t *values, size_t values_count, size_t i, uint8_t *insts, size_t
out_count)
{
    result[out_count].COM = "MOV";
    result[out_count].ARGS.hasReg1 = true;
    result[out_count].ARGS.hasReg2 = true;
    add_bits(result, out_count, insts[0]);
    add_bits(result, out_count, insts[1]);
    if (i + 1 < values_count)
    {
        add_bits(result, out_count, values[i]);
        add_bits(result, out_count, values[i + 1]);
        result[out_count].ARGS.reg1 = values[i];
        result[out_count].ARGS.reg2 = values[i + 1];
        i++;
    }
    else
    {
        result[out_count].hasError = true;
        result[out_count].ErrorMessage = "Not enough args";
    }
    return i;
}

size_t MOVREGADDR(OUT *result, uint8_t *values, size_t values_count, size_t i, uint8_t *insts,
DSC *table, size_t out_count)
{
    result[out_count].COM = "MOV";
    result[out_count].ARGS.hasReg1 = true;
    result[out_count].ARGS.hasAddr = true;
    add_bits(result, out_count, insts[0]);
    add_bits(result, out_count, insts[1]);
    if (values[i] == '0')
    {
        add_bits(result, out_count, values[i]);
        if (i + 9 < values_count)
        {
            add_bits(result, out_count, values[i + 1]);
            char for_phys[9];
            for (size_t k = 0; k < 8; k++)
```

```
            {
                add_bits(result, out_count, values[i + k + 2]);
                for_phys[k] = (char)values[i + k + 2];
            }
            result[out_count].ARGS.reg1 = values[i + 1];
            for_phys[8] = '\0';
            i += 9;
            set_vaddr(result, out_count, for_phys);
            char *physCheck = checkPhysAddr(for_phys, table);
            if (strcmp(physCheck, "") == 0)
                result[out_count].ARGS.addr = getPhysAddr(for_phys, table);
            else
            {
                result[out_count].hasError = true;
                result[out_count].ErrorMessage = physCheck;
            }
        }
        else
        {
            result[out_count].hasError = true;
            result[out_count].ErrorMessage = "Not enough args";
        }
    }
    else
    {
        result[out_count].hasError = true;
        result[out_count].ErrorMessage = "Missing 0 after 1B in MOV REG, ADDR";
    }
    return i;
}

size_t ADDREGREG(OUT *result, uint8_t *values, size_t values_count, size_t i, uint8_t *insts, size_t
out_count)
{
    result[out_count].COM = "ADD";
    result[out_count].ARGS.hasReg1 = true;
    result[out_count].ARGS.hasReg2 = true;
    add_bits(result, out_count, insts[0]);
    add_bits(result, out_count, insts[1]);
    if (i + 1 < values_count)
    {
        add_bits(result, out_count, values[i]);
        add_bits(result, out_count, values[i + 1]);
        result[out_count].ARGS.reg1 = values[i];
        result[out_count].ARGS.reg2 = values[i + 1];
        i++;
    }
    else
    {
```

```
        result[out_count].hasError = true;
        result[out_count].ErrorMessage = "Not enough args";
    }
    return i;
}

size_t ADDREGADDR(OUT *result, uint8_t *values, size_t values_count, size_t i, uint8_t *insts,
DSC *table, size_t out_count)
{
    result[out_count].COM = "ADD";
    result[out_count].ARGS.hasReg1 = true;
    result[out_count].ARGS.hasAddr = true;
    add_bits(result, out_count, insts[0]);
    add_bits(result, out_count, insts[1]);

    if (values[i] == '0')
    {
        add_bits(result, out_count, values[i]);
        if (i + 9 < values_count)
        {
            add_bits(result, out_count, values[i + 1]);
            char for_phys[9];
            for (size_t k = 0; k < 8; k++)
            {
                add_bits(result, out_count, values[i + k + 2]);
                for_phys[k] = (char)values[i + k + 2];
            }
            result[out_count].ARGS.reg1 = values[i + 1];
            for_phys[8] = '\0';
            i += 9;
            set_vaddr(result, out_count, for_phys);
            char *physCheck = checkPhysAddr(for_phys, table);
            if (strcmp(physCheck, "") == 0)
                result[out_count].ARGS.addr = getPhysAddr(for_phys, table);
            else
            {
                result[out_count].hasError = true;
                result[out_count].ErrorMessage = physCheck;
            }
        }
        else
        {
            result[out_count].hasError = true;
            result[out_count].ErrorMessage = "Not enough args";
        }
    }
    else
    {
        result[out_count].hasError = true;
```

```
        result[out_count].ErrorMessage = "Missing 0 after 02 in ADD REG, ADDR";
    }
    return i;
}

size_t JMPADDR(OUT *result, uint8_t *values, size_t values_count, size_t i, uint8_t *insts, DSC
*table, size_t out_count)
{
    result[out_count].COM = "JMP";
    result[out_count].ARGS.hasAddr = true;
    add_bits(result, out_count, insts[0]);
    add_bits(result, out_count, insts[1]);
    if (i + 7 < values_count)
    {
        char for_phys[9];
        for (size_t k = 0; k < 8; k++)
        {
            add_bits(result, out_count, values[i + k]);
            for_phys[k] = (char)values[i + k];
        }
        result[out_count].ARGS.reg1 = values[i];
        for_phys[8] = '\0';
        i += 7;
        set_vaddr(result, out_count, for_phys);
        char *physCheck = checkPhysAddr(for_phys, table);
        if (strcmp(physCheck, "") == 0)
            result[out_count].ARGS.addr = getPhysAddr(for_phys, table);
        else
        {
            result[out_count].hasError = true;
            result[out_count].ErrorMessage = physCheck;
        }
    }
    else
    {
        result[out_count].hasError = true;
        result[out_count].ErrorMessage = "Not enough args";
    }
    return i;
}

size_t JLADDR(OUT *result, uint8_t *values, size_t values_count, size_t i, uint8_t *insts, DSC *table,
size_t out_count)
{
    result[out_count].COM = "JL";
    result[out_count].ARGS.hasAddr = true;
    add_bits(result, out_count, insts[0]);
    add_bits(result, out_count, insts[1]);
    if (i + 7 < values_count)
```

```
    {
        char for_phys[9];
        for (size_t k = 0; k < 8; k++)
        {
            add_bits(result, out_count, values[i + k]);
            for_phys[k] = (char)values[i + k];
        }
        result[out_count].ARGS.reg1 = values[i];
        for_phys[8] = '\0';
        i += 7;
        set_vaddr(result, out_count, for_phys);
        char *physCheck = checkPhysAddr(for_phys, table);
        if (strcmp(physCheck, "") == 0)
            result[out_count].ARGS.addr = getPhysAddr(for_phys, table);
        else
        {
            result[out_count].hasError = true;
            result[out_count].ErrorMessage = physCheck;
        }
    }
    else
    {
        result[out_count].hasError = true;
        result[out_count].ErrorMessage = "Not enough args";
    }
    return i;
}

size_t JGADDR(OUT *result, uint8_t *values, size_t values_count, size_t i, uint8_t *insts, DSC *table,
size_t out_count)
{
    result[out_count].COM = "JG";
    result[out_count].ARGS.hasAddr = true;
    add_bits(result, out_count, insts[0]);
    add_bits(result, out_count, insts[1]);
    if (i + 7 < values_count)
    {
        char for_phys[9];
        for (size_t k = 0; k < 8; k++)
        {
            add_bits(result, out_count, values[i + k]);
            for_phys[k] = (char)values[i + k];
        }
        result[out_count].ARGS.reg1 = values[i];
        for_phys[8] = '\0';
        i += 7;
        set_vaddr(result, out_count, for_phys);
        char *physCheck = checkPhysAddr(for_phys, table);
        if (strcmp(physCheck, "") == 0)
```

```
            result[out_count].ARGS.addr = getPhysAddr(for_phys, table);
        else
        {
            result[out_count].hasError = true;
            result[out_count].ErrorMessage = physCheck;
        }
    }
    else
    {
        result[out_count].hasError = true;
        result[out_count].ErrorMessage = "Not enough args";
    }
    return i;
}

size_t CMPREGREG(OUT *result, uint8_t *values, size_t values_count, size_t i, uint8_t *insts, size_t
out_count)
{
    result[out_count].COM = "CMP";
    result[out_count].ARGS.hasReg1 = true;
    result[out_count].ARGS.hasReg2 = true;
    add_bits(result, out_count, insts[0]);
    add_bits(result, out_count, insts[1]);
    if (i + 1 < values_count)
    {
        add_bits(result, out_count, values[i]);
        add_bits(result, out_count, values[i + 1]);
        result[out_count].ARGS.reg1 = values[i];
        result[out_count].ARGS.reg2 = values[i + 1];
        i++;
    }
    else
    {
        result[out_count].hasError = true;
        result[out_count].ErrorMessage = "Not enough args";
    }
    return i;
}

size_t MOVREGLIT32(OUT *result, uint8_t *values, size_t values_count, size_t i, uint8_t *insts,
size_t out_count)
{
    result[out_count].COM = "MOV";
    result[out_count].ARGS.hasReg1 = true;
    result[out_count].ARGS.hasLit32 = true;
    add_bits(result, out_count, insts[0]);
    add_bits(result, out_count, insts[1]);
    if (values[i] == '2')
    {
```

```
            add_bits(result, out_count, values[i]);
            if (i + 9 < values_count)
            {
                add_bits(result, out_count, values[i + 1]);
                char for_lit[9];
                for (size_t k = 0; k < 8; k++)
                {
                    add_bits(result, out_count, values[i + k + 2]);
                    for_lit[k] = (char)values[i + k + 2];
                }
                result[out_count].ARGS.reg1 = values[i + 1];
                for_lit[8] = '\0';
                i += 9;
                result[out_count].ARGS.lit32 = (uint32_t)strtoll(for_lit, NULL, 16);
            }
            else
            {
                result[out_count].hasError = true;
                result[out_count].ErrorMessage = "Not enough args";
            }
        }
        else
        {
            result[out_count].hasError = true;
            result[out_count].ErrorMessage = "Missing 2 after 1C in MOV REG, LIT32";
        }
        return i;
    }
    ==> converter-utils.c <==
    #include "converter.h"
    #include <stdlib.h>
    #include <string.h>
    #include <ctype.h>

    size_t SIZE_RESULT = 500;
    size_t SIZE_VALUES = 2000;

    void add_bits(OUT *result, size_t out_c, uint8_t value)
    {
        result[out_c].values_count++;
        result[out_c].values = (uint8_t *)realloc(result[out_c].values, result[out_c].values_count *
    sizeof(uint8_t));
        if (result[out_c].values == NULL)
            exit(EXIT_FAILURE);
        result[out_c].values[result[out_c].values_count - 1] = value;
    }

    void skip(FILE *reader)
    {
```

```
    char c;
    do
        c = (char)fgetc(reader);
    while (!(c == EOF || c == ';'));
}

void UNKNOWN(OUT *result, uint8_t *insts, size_t out_c)
{
    add_bits(result, out_c, insts[0]);
    add_bits(result, out_c, insts[1]);
    result[out_c].hasError = true;
    result[out_c].ErrorMessage = "Unknown instruction";
}

OUT *realloc_with_memset_zero_out(OUT *p1)
{
    OUT *p2 = (OUT *)malloc(SIZE_RESULT * 2*sizeof(OUT));
    if (p2 == NULL)
        exit(EXIT_FAILURE);

    memset(p2, 0, SIZE_RESULT*2*sizeof(p2[0]));
    if (p1 != NULL)
    {
        for (size_t i = 0; i < SIZE_RESULT; i++)
            p2[i] = p1[i];

        SIZE_RESULT *= 2;
    }
    return p2;
}

uint8_t *realloc_with_memset_zero_uint8(uint8_t *p1)
{
    uint8_t *p2 = (uint8_t *)malloc(SIZE_VALUES * 2*sizeof(uint8_t));
    if (p2 == NULL)
        exit(EXIT_FAILURE);

    memset(p2, 0, SIZE_VALUES*2*sizeof(p2[0]));
    if (p1 != NULL)
    {
        for (size_t i = 0; i < SIZE_VALUES; i++)
            p2[i] = p1[i];

        SIZE_VALUES *= 2;
    }
    return p2;
}

valRet create_values(char *_input)
```

```
{
    size_t values_count = 0;
    FILE *reader = fopen(_input, "r");
    if (reader == NULL)
        exit(-1);
    uint8_t *values = NULL;
    values = realloc_with_memset_zero_uint8(values);
    char c = '\0';
    do
    {
        c = (char)fgetc(reader);
        if (c == ';')
            skip(reader);
        else
        {
            if (isalnum(c))
            {
                values[values_count] = (uint8_t)c;
                values_count++;

                if (values_count == SIZE_VALUES)
                    values = realloc_with_memset_zero_uint8(values);
            }
        }
    } while (c != EOF);
    fclose(reader);
    valRet ret = {values_count, values};
    return ret;
}
==> converter.c <==
#include "converter.h"
#include <stdlib.h>
#include <stdbool.h>
#include <ctype.h>
#include <string.h>

outRet convert(char *_input, DSC *table)
{
    size_t out_count = 0;
    valRet valStruct = create_values(_input);
    uint8_t* values = valStruct.values;
    size_t values_count = valStruct.count;
    uint8_t inst_i = 0;
    uint8_t insts[2];
    OUT *result = NULL;
    result = realloc_with_memset_zero_out(result);
    for (size_t i = 0; i < values_count; i++)
    {
        if (inst_i < 2)
```

```
    insts[inst_i++] = values[i];
else
{
    char instruction[2] = {(char)insts[0], (char)insts[1]};
    uint8_t command = (uint8_t)strtol(instruction, NULL, 16);
    switch (command)
    {
    case 0x1A:
        // MOV REG1,REG2
        i = mov1a();
        break;
    case 0x1B:
        // MOV REG, ADDR
        i = mov1b();
        break;
    case 0x01:
        // ADD REG, REG
        i = add01();
        break;
    case 0x02:
        // ADD REG, ADDR
        i = add02();
        break;
    case 0x91:
        // JMP ADDR
        i = jmp91();
        break;
    case 0x93:
        // JL ADDR
        i = jl93();
        break;
    case 0x95:
        // JG ADDR
        i = jg95();
        break;
    case 0x80:
        // CMP REG, REG
        i = cmp80();
        break;
    case 0x1C:
        // MOV REG, LIT32
        i = mov1c();
        break;
    default:
        UNKNOWN(result, insts, out_count);
        break;
    }
    inst_i = 0;
    out_count++;
```

```
            if (out_count == SIZE_RESULT)
                result = realloc_with_memset_zero_out(result);
        }
    }

    if (inst_i != 0)
    {
        result[out_count].hasError = true;
        result[out_count].ErrorMessage = "Readen instruction, but no operands";
        add_bits(result, out_count, insts[0]);
        if (inst_i == 2)
            add_bits(result, out_count, insts[1]);
        out_count++;
    }

    free(values);
    outRet ret = {out_count,result};
    return ret;
}
==> main.c <==
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <stdbool.h>
#include "cli.h"
#include "outputting.h"
#include "verify.h"

int main(int argc, char *argv[])
{
    Files my_files = {NULL, NULL, NULL, NULL, true};
    /* processing cli */
    if (proc_cli(argc, argv, &my_files))
    {
        /* starting work */
        if (my_files.verbose)
            printf("Start working\n");

        /* parsing table from CSV file */
        DSC *my_table = parseTable(my_files._table);
        if (my_files.verbose)
            printf("Table parsing is done\n");

        /* converting bytes to instructions, virtual addresses to physical*/
        outRet convStruct = convert(my_files._input, my_table);
        OUT *converted = convStruct.result;
        size_t out_count = convStruct.count;

        /* verbose output if needed */
```

```
        if (my_files.verbose)
            printOut(out_count, converted);

        /* output to file */
        writeOut(out_count, converted, my_files._output);

        /* freeing memory */
        free(converted->values);
        free(converted->ARGS.vaddr);
        free(converted);
        free(my_table);

        /* finished work */
        if (my_files.verbose)
            printf("Finished work\n");

        /* verifying work */
        if (my_files._verify != NULL)
        {
            verifyRet verified = verify(my_files._output, my_files._verify);
            if (!verified.got_error)
                printf("Verification passed\n");
            else
            {
                for (size_t i = 0; i < verified.error_count; i++)
                    printf("%s\n", verified.error[i]);
            }
        }
    }
    return 0;
}
==> physical_addr.c <==
#include "physical_addr.h"
#include <stdlib.h>

char *checkPhysAddr(char *for_phys, DSC *table)
{
    uint32_t number = (uint32_t)strtoll(for_phys, NULL, 16);
    uint16_t num_of_d = (uint16_t)(number >> 21);
    uint32_t mask = 0x1fffff;
    uint32_t offset = (number & mask);

    if (num_of_d < RDT)
    {
        if(table[num_of_d].segment_size != 0){
        if (table[num_of_d].in_memory)
        {
            uint32_t physical = table[num_of_d].base_addr + offset;
            if (physical < table[num_of_d].base_addr + table[num_of_d].segment_size)
```

```
            {
                return "";
            }
            else
            {
                return "Out of segment";
            }
        }
        else
        {
            return "Flag not in memory";
        }
        }
        else
        {
            return "Segment does not exist";
        }
    }
    else
    {
        return "Number of segment > RDT";
    }
}

uint32_t getPhysAddr(char *for_phys, DSC *table)
{
    uint32_t number = (uint32_t)strtoll(for_phys, NULL, 16);
    uint16_t num_of_d = (uint16_t)(number >> 21);
    uint32_t mask = 0x1fffff;
    uint32_t offset = (number & mask);

    uint32_t physical = table[num_of_d].base_addr + offset;
    return physical;
}
==> print_output.c <==
#include "outputting.h"

void printOut(size_t out_count, OUT* converted)
{
        printf("Results output created\n");
    printf("----------------------\n");
    for (size_t i = 0; i < out_count; i++)
    {
        int p = 0;
        for (size_t k = 0; k < converted[i].values_count; k++)
        {
            printf("%c", (char)converted[i].values[k]);
            p++;
            if (p == 2)
```

```
    {
        printf(" ");
        p = 0;
    }
}
printf("\n");
if (converted[i].hasError)
{
    printf("%s\n", converted[i].ErrorMessage);
    if (converted[i].ARGS.hasAddr)
    {
        printf("Virtual address:%s\n", converted[i].ARGS.vaddr);
        printf("Seg id: 0x%" PRIx16 "\n", converted[i].ARGS.seg);
        printf("Seg offset: 0x%" PRIx32 "\n", converted[i].ARGS.offset);
    }
}
else
{
    printf("%s ", converted[i].COM);
    if (converted[i].ARGS.hasReg1)
    {
        char val[2] = {(char)converted[i].ARGS.reg1, '\0'};
        printf("R%lu", strtol(val, NULL, 16));
    }
    if (converted[i].ARGS.hasReg2)
    {
        char val[2] = {(char)converted[i].ARGS.reg2, '\0'};
        printf(", R%lu", strtol(val, NULL, 16));
    }
    if (converted[i].ARGS.hasLit32)
    {
        char buff[100];
        buff[0] = '0';
        snprintf(buff, 100, "%" PRIx32, converted[i].ARGS.lit32);
        if (isalpha(buff[0]))
            printf(", 0");
        else
            printf(", ");
        printf("%" PRIx32 "h", converted[i].ARGS.lit32);
    }
    if (converted[i].ARGS.hasAddr)
    {
        if (converted[i].ARGS.hasReg1)
        {
            printf(", ");
        }
        printf("[0x%" PRIx32 "]", converted[i].ARGS.addr);
    }
    printf("\n");
```

```
        }
        printf("----------------------\n");
    }
}
==> table.c <==
#include "table.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAXCHAR 100


DSC* parseTable(char* _table)
{
    DSC* Table;
    Table = (DSC*)malloc(RDT*sizeof(DSC));
    memset(Table,0,RDT*sizeof(Table[0]));
    FILE* reader = fopen(_table, "r");
    if(reader == NULL)
        exit(-1);

    char row[MAXCHAR];
    char *token;
    while(feof(reader) != true)
    {
        fgets(row, MAXCHAR, reader);
        token  = strtok(row, "|");
        size_t id = (size_t)atoi(token);
        size_t parse = 0;
        while(token != NULL)
        {
            token = strtok(NULL, "|");
            switch(parse)
            {
                case 0:
                    Table[id].base_addr = (uint32_t)strtoul(token, NULL, 0);
                    break;
                case 1:
                    Table[id].segment_size = (uint16_t)atoi(token);
                    break;
                case 2:
                    Table[id].in_memory = (bool)atoi(token);
                    break;
                default:
                    break;
            };
            parse++;
        }
    }
```

```
    fclose(reader);
    return Table;
}
==> verify.c <==
#include "verify.h"
#include <stdio.h>
#include <ctype.h>
#include <stdbool.h>
char c = '\0';
char d = '\0';
size_t row = 0;
size_t col = 0;
size_t v_row = 0;
size_t v_col = 0;

void update_row_col(bool is_c)
{
    if (is_c)
    {
        if (c == '\n')
        {
            row++;
            col = 0;
        }
        else
            col++;
    }
    else
    {
        if (d == '\n')
        {
            v_row++;
            v_col = 0;
        }
        else
            v_col++;
    }
}

void cleanCom(FILE *read_output, FILE *read_verify)
{
    if (d == ';' || c == ';')
    {
        if (c == ';')
        {
            do
            {
                c = (char)getc(read_output);
                update_row_col(true);
```

```
            } while (c != EOF && c != ';');
            c = (char)getc(read_output);
            update_row_col(true);
            if (c == '\n'){
                c = (char)getc(read_output);
                update_row_col(true);
            }
        }
        if (d == ';')
        {
            do
            {
                d = (char)getc(read_verify);
                update_row_col(false);
            } while (d != EOF && d != ';');
            d = (char)getc(read_verify);
            update_row_col(false);
            if (d == '\n'){
                d = (char)getc(read_verify);
                update_row_col(false);
            }
        }
    }
    if (d == ';' || c == ';')
        cleanCom(read_output, read_verify);
}
verifyRet verify(char *_output, char *_verify)
{
    FILE *read_output = fopen(_output, "r");
    FILE *read_verify = fopen(_verify, "r");
    verifyRet ret = {.got_error = false, .error_count = 0};
    do
    {
        do
        {
            c = (char)getc(read_output);
            update_row_col(true);
        } while (c != EOF && !(isalnum(c) || c == ';' || c == ','));
        do
        {
            d = (char)getc(read_verify);
            update_row_col(false);
        } while (d != EOF && !(isalnum(d) || d == ';' || d == ','));

        cleanCom(read_output, read_verify);
        if (c != d)
        {
            ret.error[ret.error_count] = malloc(ERROR_SIZE * sizeof(char));
```

```
            snprintf(ret.error[ret.error_count++], ERROR_SIZE, "Output(%llu:%llu:%c) != Verify(%llu:
%llu:%c), but have to", row + 1, col, c, v_row + 1, v_col, d);
            ret.got_error = true;
            if (ret.error_count == DEEP)
                return ret;
        }

    } while (!(c == EOF || d == EOF));
    return ret;
}
==> virtual_addr.c <==
#include "virtual_addr.h"
#include <stdlib.h>
void set_vaddr(OUT *result, size_t out_c, char* for_phys)
{
    result[out_c].ARGS.vaddr = malloc(13*sizeof(char));
    size_t k = 0;
    size_t space = 0;
    for(size_t i = 0; i < 12; i++){
        if(space != 2){
        result[out_c].ARGS.vaddr[i] = for_phys[k];
        k++;
        space++;
        }
        else
        {
            space = 0;
            result[out_c].ARGS.vaddr[i] = ' ';
        }
    }
    result[out_c].ARGS.vaddr[12] = '\0';
    uint32_t number = (uint32_t)strtol(for_phys, NULL, 16);
    result[out_c].ARGS.seg = (uint16_t)(number >> 21);
    uint32_t mask = 0x1fffff;
    result[out_c].ARGS.offset = (number & mask);
}
==> write_output.c <==
#include "outputting.h"

void writeOut(size_t out_count, OUT *converted, char *_output)
{
    FILE *out_to = fopen(_output, "w");
    if (out_to == NULL)
        exit(-1);
    for (size_t i = 0; i < out_count; i++)
    {
        int p = 0;
        for (size_t k = 0; k < converted[i].values_count; k++)
        {
```

```
            fprintf(out_to, "%c", (char)converted[i].values[k]);
            p++;
            if (p == 2)
            {
                fprintf(out_to, " ");
                p = 0;
            }
        }
        fprintf(out_to, "\n");
        if (converted[i].hasError)
            fprintf(out_to, "%s\n", converted[i].ErrorMessage);
        else
        {
            fprintf(out_to, "%s ", converted[i].COM);
            if (converted[i].ARGS.hasReg1)
            {
                char val[2] = {(char)converted[i].ARGS.reg1, '\0'};
                fprintf(out_to, "R%lu", strtol(val, NULL, 16));
            }
            if (converted[i].ARGS.hasReg2)
            {
                char val[2] = {(char)converted[i].ARGS.reg2, '\0'};
                fprintf(out_to, ", R%lu", strtol(val, NULL, 16));
            }
            if (converted[i].ARGS.hasLit32)
            {
                char buff[100];
                buff[0] = '0';
                snprintf(buff, 100, "%" PRIx32, converted[i].ARGS.lit32);
                if (isalpha(buff[0]))
                    fprintf(out_to, ", 0");
                else
                    fprintf(out_to, ", ");
                fprintf(out_to, "%" PRIx32 "h", converted[i].ARGS.lit32);
            }
            if (converted[i].ARGS.hasAddr)
            {
                if (converted[i].ARGS.hasReg1)
                    fprintf(out_to, ", ");
                fprintf(out_to, "[0x%" PRIx32 "]", converted[i].ARGS.addr);
            }
            fprintf(out_to, "\n");
        }
    }
    fclose(out_to);
}
==> cli.h <==
#include "file_dest.h"
#ifndef CLI_H
```

```
#define CLI_H
bool proc_cli(int argc, char *argv[],Files* my_files);

#endif
==> converter.h <==
#include "physical_addr.h"
#include "virtual_addr.h"
#include "values_return_struct.h"
#include <stdio.h>

#ifndef CONVERTER_H
#define CONVERTER_h

/* converter */
outRet convert(char *_input, DSC *table);

/* converter-utils */
extern size_t SIZE_RESULT;
extern size_t SIZE_VALUES;

void add_bits(OUT *result, size_t out_c, uint8_t value);
void skip(FILE *reader);
void UNKNOWN(OUT *result, uint8_t *insts, size_t out_c);
OUT *realloc_with_memset_zero_out(OUT *p1);
uint8_t *realloc_with_memset_zero_uint8(uint8_t *p1);
valRet create_values(char *_input);

/* converter-commands */

size_t MOVREGREG(OUT *result, uint8_t *values, size_t values_count, size_t i, uint8_t *insts, size_t
out_count);
size_t MOVREGADDR(OUT *result, uint8_t *values, size_t values_count, size_t i, uint8_t *insts,
DSC *table, size_t out_count);
size_t ADDREGREG(OUT *result, uint8_t *values, size_t values_count, size_t i, uint8_t *insts, size_t
out_count);
size_t ADDREGADDR(OUT *result, uint8_t *values, size_t values_count, size_t i, uint8_t *insts,
DSC *table, size_t out_count);
size_t JMPADDR(OUT *result, uint8_t *values, size_t values_count, size_t i, uint8_t *insts, DSC
*table, size_t out_count);
size_t JLADDR(OUT *result, uint8_t *values, size_t values_count, size_t i, uint8_t *insts, DSC *table,
size_t out_count);
size_t JGADDR(OUT *result, uint8_t *values, size_t values_count, size_t i, uint8_t *insts, DSC *table,
size_t out_count);
size_t CMPREGREG(OUT *result, uint8_t *values, size_t values_count, size_t i, uint8_t *insts, size_t
out_count);
size_t MOVREGLIT32(OUT *result, uint8_t *values, size_t values_count, size_t i, uint8_t *insts,
size_t out_count);

/* converter-commands-macro*/
```

```
#define mov1a() MOVREGREG(result, values, values_count, i, insts, out_count)
#define mov1b() MOVREGADDR(result, values, values_count, i, insts, table, out_count)
#define add01() ADDREGREG(result, values, values_count, i, insts, out_count)
#define add02() ADDREGADDR(result, values, values_count, i, insts, table, out_count)
#define jmp91() JMPADDR(result, values, values_count, i, insts, table, out_count)
#define jl93() JLADDR(result, values, values_count, i, insts, table, out_count)
#define jg95() JGADDR(result, values, values_count, i, insts, table, out_count)
#define cmp80() CMPREGREG(result, values, values_count, i, insts, out_count)
#define mov1c() MOVREGLIT32(result, values, values_count, i, insts, out_count)

#endif
==> file_dest.h <==
#include <stdbool.h>
#ifndef FILE_DEST_H
#define FILE_DEST_H

struct files{
    char* _input;
    char* _output;
    char* _table;
    char* _verify;
    bool verbose;
};
typedef struct files Files;

#endif
==> out_return_struct.h <==
#include <stdint.h>
#include <stdlib.h>
#include "output_struct.h"
#ifndef OUT_RETURN_STRUCT_H
#define OUT_RETURN_STRUCT_H

struct out_return{
    size_t count;
    OUT* result;
};
typedef struct out_return outRet;

#endif
==> output_struct.h <==
#include <stdint.h>
#include <stdbool.h>
#ifndef OUTPUT_STRUCT
#define OUTPUT_STRUCT

struct args{
    bool hasReg1;
```

```
    bool hasReg2;
    bool hasAddr;
    bool hasLit32;
    uint8_t reg1;
    uint8_t reg2;
    uint32_t addr;
    char* vaddr;
    uint16_t seg;
    uint32_t offset;
    uint32_t lit32;
};


struct output
{
    uint8_t* values;
    size_t values_count;
    char* COM;
    struct args ARGS;
    char* ErrorMessage;
    bool hasError;
};
typedef struct output OUT;

#endif
==> outputting.h <==
#include "converter.h"
#include <inttypes.h>
#include <ctype.h>
#ifndef OUTPUTTING_H
#define OUTPUTTING_H

void printOut(size_t out_count, OUT* converted);
void writeOut(size_t out_count, OUT* converted, char* _output);

#endif
==> physical_addr.h <==
#include "table.h"
#ifndef PHYSICAL_ADDR
#define PHYSICAL_ADDR

char *checkPhysAddr(char *for_phys, DSC *table);
uint32_t getPhysAddr(char *for_phys, DSC *table);


#endif
==> table.h <==
#include <stdint.h>
#include <stdbool.h>
```

```
#ifndef TABLE_H
#define TABLE_H
#define RDT 2048

struct descriptor
{
    uint32_t base_addr;
    uint16_t segment_size;
    bool in_memory;
    bool other[11];
};
typedef struct descriptor DSC;

DSC* parseTable(char* _table);

#endif
==> values_return_struct.h <==
#include <stdint.h>
#include <stdlib.h>
#ifndef VALUES_RETURN_STRUCT_H
#define VALUES_RETURN_STRUCT_H

struct values_return{
    size_t count;
    uint8_t* values;
};
typedef struct values_return valRet;

#endif
==> verify.h <==
#ifndef VERIFY_H
#define VERIFY_H
#include <stdbool.h>
#include <stdlib.h>

#define DEEP 3
#define ERROR_SIZE 200

struct verify_return
{
    bool got_error;
    char* error[DEEP];
    size_t error_count;
};
typedef struct verify_return verifyRet;

verifyRet verify(char *_output, char *_verify);

#endif
```

==> virtual_addr.h <==

```
#include "out_return_struct.h"
#ifndef VIRTUAL_ADDR
#define VIRTUAL_ADDR

void set_vaddr(OUT *result, size_t out_c, char* for_phys);

#endif
```

Тестування

```
C:\Users\t3\arch-lab3>make test
outDebug.exe  -i ./tests/add01/input.txt -o ./tests/add01/output.txt -t ./tests/add01/memory.csv -v ./tests/add01/verify.txt -q
Output will be verified
Verification passed
outDebug.exe  -i ./tests/add02/input.txt -o ./tests/add02/output.txt -t ./tests/add02/memory.csv -v ./tests/add02/verify.txt -q
Output will be verified
Verification passed
outDebug.exe  -i ./tests/cmp80/input.txt -o ./tests/cmp80/output.txt -t ./tests/cmp80/memory.csv -v ./tests/cmp80/verify.txt -q
Output will be verified
Verification passed
outDebug.exe  -i ./tests/jg95/input.txt -o ./tests/jg95/output.txt -t ./tests/jg95/memory.csv -v ./tests/jg95/verify.txt -q
Output will be verified
Verification passed
outDebug.exe  -i ./tests/jl93/input.txt -o ./tests/jl93/output.txt -t ./tests/jl93/memory.csv -v ./tests/jl93/verify.txt -q
Output will be verified
Verification passed
outDebug.exe  -i ./tests/jmp91/input.txt -o ./tests/jmp91/output.txt -t ./tests/jmp91/memory.csv -v ./tests/jmp91/verify.txt -q
Output will be verified
Verification passed
outDebug.exe  -i ./tests/mov1a/input.txt -o ./tests/mov1a/output.txt -t ./tests/mov1a/memory.csv -v ./tests/mov1a/verify.txt -q
Output will be verified
Verification passed
outDebug.exe  -i ./tests/mov1b/input.txt -o ./tests/mov1b/output.txt -t ./tests/mov1b/memory.csv -v ./tests/mov1b/verify.txt -q
Output will be verified
Verification passed
outDebug.exe  -i ./tests/mov1c/input.txt -o ./tests/mov1c/output.txt -t ./tests/mov1c/memory.csv -v ./tests/mov1c/verify.txt -q
Output will be verified
Verification passed
```

Рис. 1 — Тестування всіх інструкцій

З метою тестування програми, для кожної інструкції(команди) був розроблений тест, що складається з вхідного файлу, файлу таблиці дескрипторів та файлу звірки. Якщо при перевірці програма помітить відмінності від файлу звірки, то виведе повідомлення про помилку та вкаже номер рядка та символ.

Рис. 2 — Приклад тесту(без використання фізичних адрес)

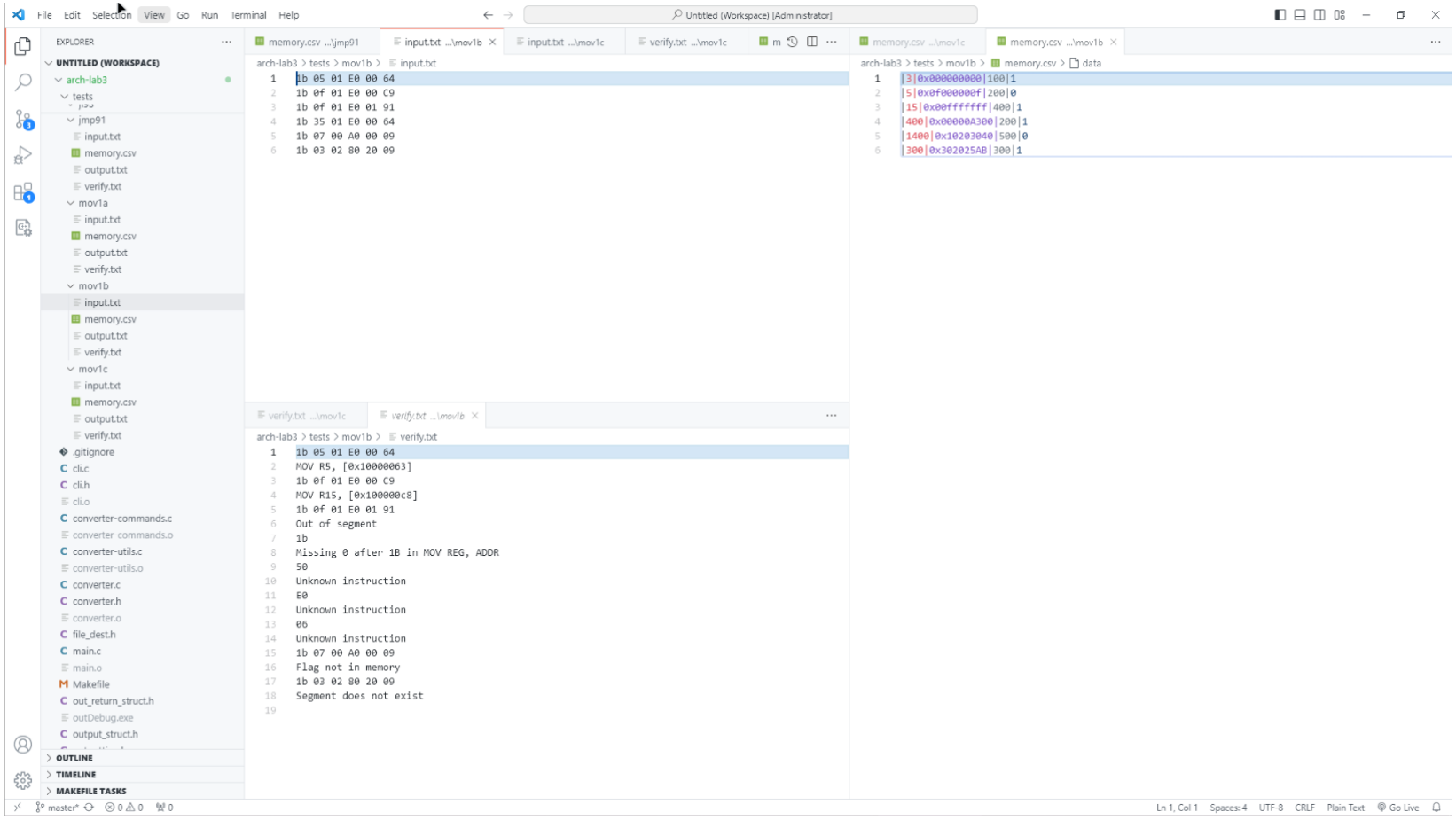

Рис. 3 — Приклад тесту(з використанням фізичної адреси)

```
PS C:\Users\t3\arch-lab3> make testjmp91
outDebug.exe  -i ./tests/jmp91/input.txt -o ./tests/jmp91/output.txt -t ./tests/jmp91/memory.csv -v ./tests/jmp91/verify.txt
Output will be verified
Start working
Table parsing is done
Results output created
----------------------
95 01 E0 00 64
JG [0x10000063]
----------------------
95 01 E0 00 C9
JG [0x100000c8]
----------------------
95 01 E0 01 91
Out of segment
Virtual address:01 E0 01 91
Seg id: 0xf
Seg offset: 0x191
----------------------
95 00 A0 00 09
Flag not in memory
Virtual address:00 A0 00 09
Seg id: 0x5
Seg offset: 0x9
----------------------
95 02 80 20 09
Segment does not exist
Virtual address:02 80 20 09
Seg id: 0x14
Seg offset: 0x2009
----------------------
Finished work
Verification passed
PS C:\Users\t3\arch-lab3>
```

Рис. 4 — Приклад роботи програми(без “quiet’’ режиму)