Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ
УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ» (ТУСУР)
Кафедра безопасности информационных систем (БИС)

ИТОГОВЫЙ ОТЧЕТ

по лабораторным работам №1-3
по дисциплине
«Системное программирование»
Вариант №12

Студент гр.739-1
_____Лобанов А.А.
__.04.2023

Руководитель
Ст. преподаватель каф. КИБЭВС
_____ _____ Пехов О.В.
__.04.2023

Томск 2023

Введение

В данном отчете собраны результаты лабораторных работ №1-3.

1 ЛАБОРАТОРНАЯ РАБОТА №1 «LINUX, DOCKER, ОКРУЖЕНИЕ ДЛЯ РАЗРАБОТКИ»

1.1 Цель работы

Ознакомиться с операционной системой и основными командами ОС Unix/Linux, ознакомиться с возможностями Docker для контейнеризации окружения программного обеспечения и его зависимостей, средств разработки и сборки ПО.

1.2 Задание

1. Ознакомиться с операционной системой и основными командами ОС Unix/Linux.

2. Ознакомиться с возможностями Docker для контейнеризации окружения программного обеспечения и его зависимостей, средств разработки и сборки ПО.

3. Вспомнить как работать с git. Создать git репозитарий.

4. Подготовить Dockerfile, в котором осуществляется виртуализация операционной системы, соответствующей варианту задания (**ОС – Alpine**), устанавливаются необходимые для выполнения задания пакеты и т.д.

5. В git залить код готовой программы и Dockerfile.

6. На moodle залить архив, содержащий: Отчет, Dockerfile, все файлы проекта.

7. Написать отчет и защитить у преподавателя. Продемонстрировать умение работать с контейнерами и образами Docker, базовыми командами ОС Linux на защите.

Программу написать в соответствии со следующим заданием: разработать скрипт для поиска файлов:

– запрашивает тип действия: поиск по имени файла или поиск по размеру;

– запрашивает каталог, в котором нужно найти файл;

– запрашивает имя файла или размер в зависимости от требуемого действия выводит все файлы с заданным именем или все файлы больше указанного размера в зависимости от требуемого действия.

Для выполнения задания использовать команду find.

1.3 Ход работы

В самом начале создания среды по выполнению задания был написан скрипт, который выполняет требования к функционалу программы.

На рисунке 1.1 представлено введение – информация о программе и ее авторе.

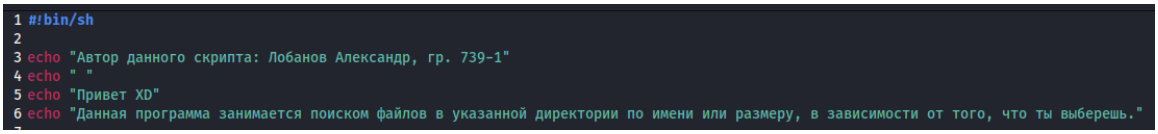

Рисунок 1.1 – Информация о программе

На рисунке 1.2 – просьба ввести свое ФИО.

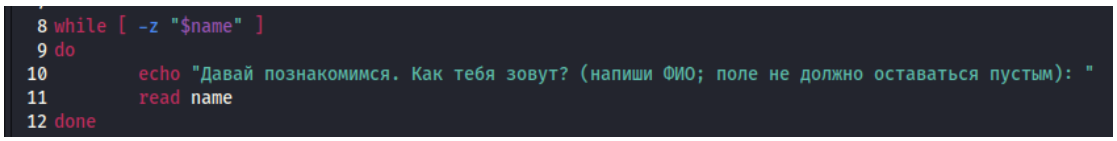

Рисунок 1.2 – Просьба ввести свое ФИО

Все программа находится в цикле WHILE. Все, что пользователь вводит с клавиатуры после просьбы программы, заносится в переменные, которые затем используются в реализации функционала. На рисунке 1.3 – выбор действия. На рисунке 1.4 – действия, если был выбран поиск по имени. На рисунке 1.5 – если по размеру файла. На рисунке 1.6 показано завершение программы – задается вопрос о желании пользователя закончить. Если ответ положительный, то цикл начинает новую итерацию с момента выбора действия. Если нет, то программа завершает свою работу.

```
while [ -z "$case" ]
do
        echo ""
        echo "Как будем искать файл: по имени (введи 1) или размеру, начиная от введенной величины (введи 2):"
        read case
        if [ "$case" -ne 1 ] && [ "$case" -ne 2 ]
        then
                echo "Неверный формат"
                case=""
                continue
        else break
        fi
done
```

Рисунок 1.3 – Выбор действия

```
if [ "$case" == 1 ]
then
        while [[ ! -d "$directory" ]]
        do
                echo "Напишите каталог, в котором будет производиться поиск (не может быть пустым или несуществующим)"
                read directory
        done

        while [ -z "$file_name" ]
        do
                echo "Напишите имя файла, который будет искаться(не может быть пустым или несуществующим)"
                read file_name
        done

        find $directory -iname $file_name.*
        echo ""
```

Рисунок 1.4 – Действия, если был выбран поиск по имени

```
else
        while [[ ! -d "$directory" ]]
        do
                echo "Напишите каталог, в котором будет производиться поиск (не может быть пустым или несуществующим)"
                read directory
        done

        while [ -z "$file_size" ]
        do
                echo "Введите минимальный размер файла (от этого числа КИЛОБАЙТ будет производится поиск файла)"
                read file_size
                if [[ ! $file_size =~ $REGEX ]]
                then
                        echo "Ввели не число"
                        file_size=""
                else
                        break
                fi
        done

        find $directory -size +"$file_size"k
        echo ""
fi
```

Рисунок 1.5 – Действия, если был выбран поиск по размеру

```
while :
do
        echo "Хотите завершить работу? (да/нет. Другие ответы не принимаются!)";
        read text
        if [ $text == "нет" ]
        then
                echo ""
                break
        elif [ $text == "да" ]
        then
                echo ""
                i=$(( $i + 1 ))
                break
        else
                echo "Написал какую-то чушь или вообще ничего не написал)";
                echo ""
        fi
done
```

Рисунок 1.6 – Завершение

Следующим шагом стояла сборка образа. Сначала был написан DockerFile, являющийся конфигурационным файлом, на основе которого и будет собран образ. На рисунке 1.7 показан данный файл.

```
1 FROM alpine:latest
2 COPY test.sh .
3 RUN chmod +x test.sh
4 RUN apk update && apk add bash && apk add --no-cache bash
5 CMD ./test.sh
6
```

Рисунок 1.7 – DockerFile

Внутри него копируется скрипт и делается исполняемым. Также в образе alpine доустанавливаются необходимые пакеты, а в CMD – команды написанного скрипта, которые будут выполнятся после старта контейнера.

Далее производилась сборка на основе DockerFile собственного образа. Все параметры, кроме CMD, были выполнены на этапе сборки (рисунок 1.8).

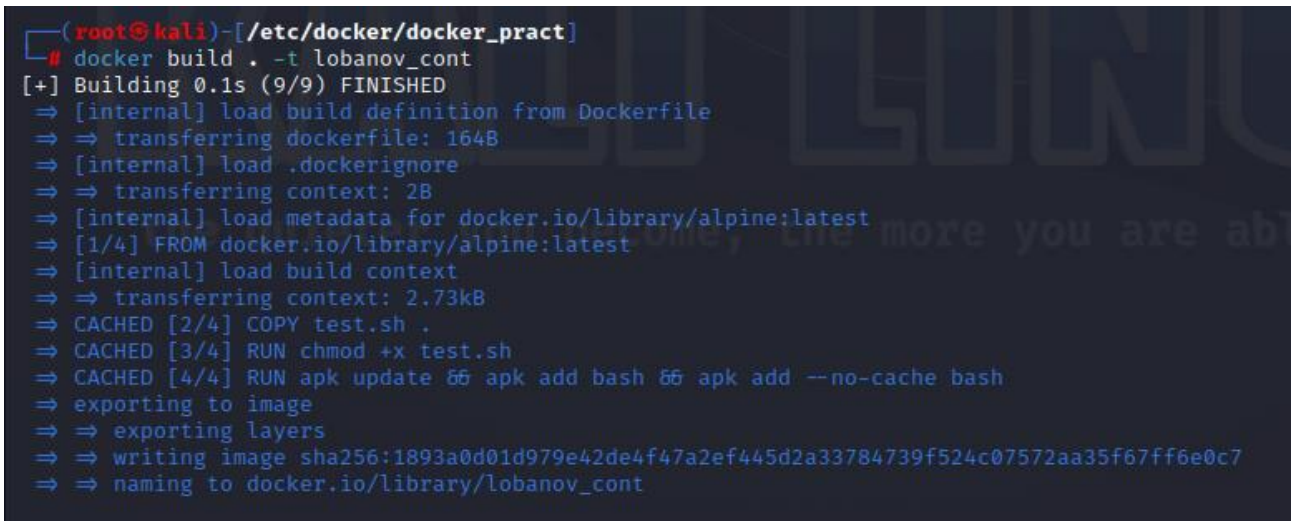

Рисунок 1.8 – Сборка образа

Ниже на рисунке 1.9 представлена команда по запуску программы в контейнере, а также исполнение самого скрипта:

1. Информация о создателе, справка о программе.
2. Ввод пользователем своего имени.
3. Выбор действия: поиска по имени или по размеру.
4. Ввод каталога, в котором будет производиться поиск.
5. Ввод имени файла, если поиск по имени.
6. Вопрос о завершении работы программы.
7. Неправильный ввод каталога.
8. Неправильный ввод ответа на вопрос о завершении работы программы.
9. Выбор поиска по размеру.
10. Ввод минимального размера файла.
11. Результат поиска.
12. Положительный ответ на вопрос о завершении, после чего программа завершает работу.

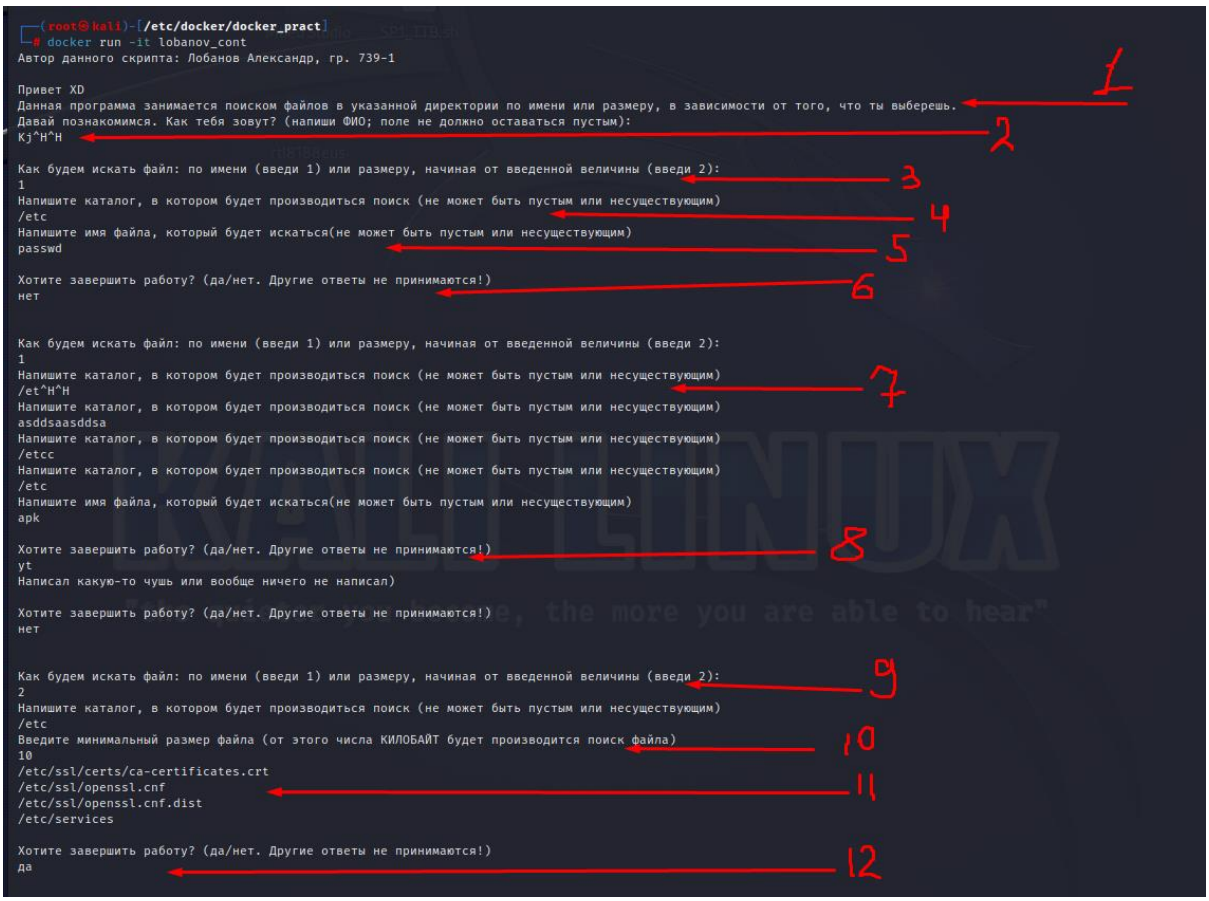

Рисунок 1.9 – Запуск контейнера

На рисунке 1.10 видно, что контейнер запущен.

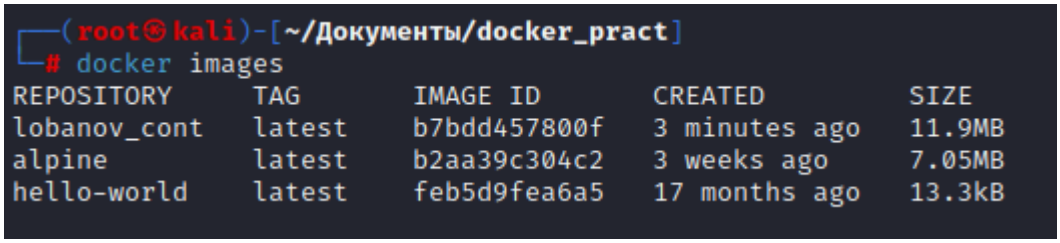

Рисунок 1.10 – Запущенные контейнеры

DockerFile и файл программы также были загружены в GitHub (рисунок 1.11), используя версию в браузере, что можно было бы сделать и в командной строке.

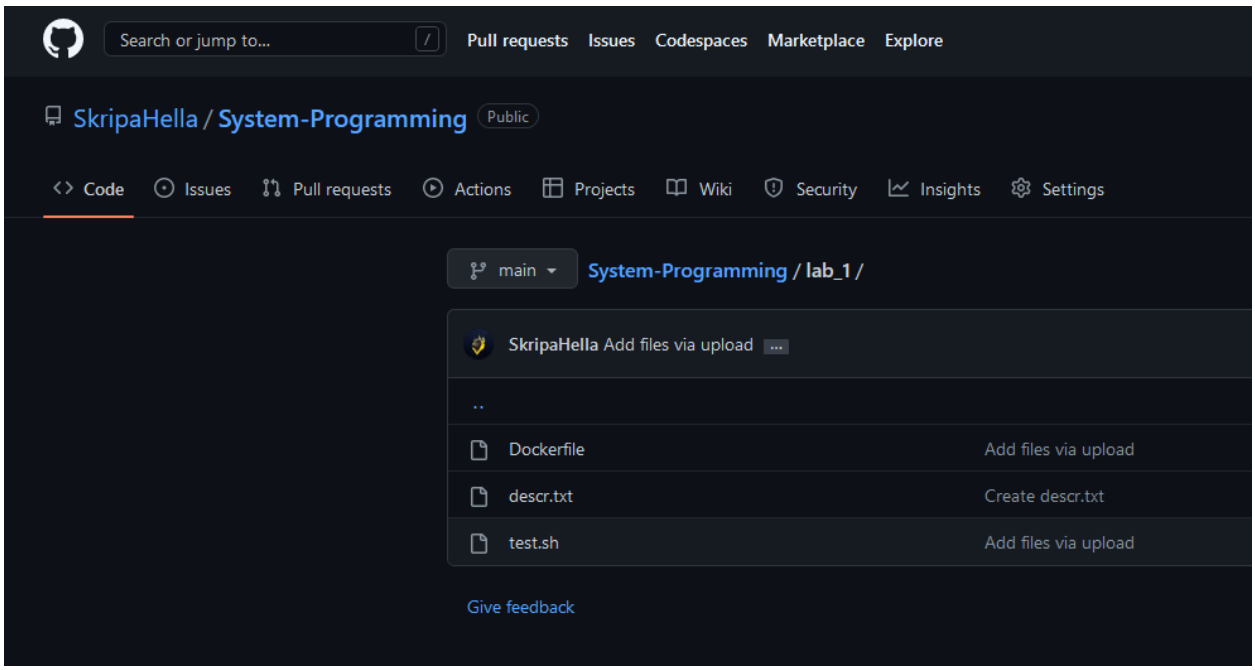

Рисунок 1.11 – Загруженные файлы в Git

Код программы представлен по ссылке в приложении А.

1.4 Заключение

В лабораторной работе №1 была освоена работа с операционной системой и основными командами ОС Unix/Linux, изучены возможности Docker для контейнеризации окружения программного обеспечения и его зависимостей, средств разработки и сборки ПО.

2 ЛАБОРАТОРНАЯ РАБОТА №2 «СИСТЕМА КОМАНД МИКРОПРОЦЕССОРОВ СЕМЕЙСТВА INTEL MCS-51»

2.1 Цель работы

Изучение принципов работы и системы команд микропроцессора на примере микропроцессоров семейства Intel mcs-51.

2.2 Задание

Составить алгоритм программы в соответствии с индивидуальным вариантом (12): установить 1 в 4-ых битах всех элементов массива из 10 байтов. Определить сумму элементов полученного массива. результат вывести на семисегментный дисплей.

2.3 Ход работы

Сначала в память были загружены значения массива (рисунок 2.1).

```
11 MOV R1, #20h ; номер ¤чейки, куда заносим
12 MOV @R1, #9Fh ; значение, которое заносим
13 MOV R1, #21h
14 MOV @R1, #66h
15 MOV R1, #22h
16 MOV @R1, #0E2h
17 MOV R1, #23h
18 MOV @R1, #0F0h
19 MOV R1, #24h
20 MOV @R1, #4Dh
21 MOV R1, #25h
22 MOV @R1, #0A5h
23 MOV R1, #26h
24 MOV @R1, #0A6h
25 MOV R1, #27h
26 MOV @R1, #4Dh
27 MOV R1, #28h
28 MOV @R1, #36h
29 MOV R1, #29h
30 MOV @R1, #0B4h
```

Рисунок 2.1 – Загрузка значений массива

Реализовывалось это с помощью косвенного адреса через R1. Сначала в регистр записывался адрес байта в памяти, затем, командой MOV @R1, в память было занесено значение, например, 9F.

Далее регистру было присвоено значение адреса на первый элемент массива и выполнен переход к основной части программы (через команду JMP one, код представлен на рисунке 2.2). JMP – команда безусловного перехода к метке, точке программы (в данном случае one). После one идет изменение в 4 бите загруженного в аккумулятор числа. Изменение на 1, не важно, была она там или нет. После этого происходит загрузка измененного числа в тот же байт, откуда оно было взято. Затем из R3 выгружается сумма предыдущих элементов массива (первое число будет складываться, соответственно, с 0). Попадают они туда после произведения операции сложения ADD и переноса результата в R3.

Следующим шагом происходит проверка на наличие бита переноса. Если при суммировании результату требуется еще один бит, то бит переноса становится равным 1, следовательно, увеличивается значение R7, который будет отвечать за старшие 4 бита, то есть старшее шестнадцатеричное число.

В R0 находится счетчик на 10 символов, который при переходе на three сравнивается с 0, и если он не равен ему, то декрементируется и выполнение описанной части программы начинается заново.

```
32 MOV R1, #20h
33 JMP one
34
35 two:
36     INC R7
37     JMP three
38
39 one:
40     MOV A, @R1
41     ORL A, #00001000b
42     MOV @R1, A
43     MOV A, R3
44     ADD A, @R1
45     MOV R3, A
46     INC R1
47     JC two
48 three:
49     DJNZ R0, one
```

Рисунок 2.2 – Программа по изменению 4 бита и суммированию элементов

Для того, чтобы результат посимвольно вывести на0020индикаторы, встала задача разместить каждые 4 бита в отдельные регистры. Для этого использовалась логическая операция AND, где старшие 4 бита результата суммы складывались с 11110000. Как видно, в результате будет выделены 4 бита, они заносятся в R6. Аналогично, с младшими, они заносятся в R5 (рисунок 2.3).

```
51 MOV A, R3
52 ANL A, #11110000b
53 SWAP A
54 MOV R6, A
55 MOV A, R3
56 ANL A, #00001111b
57 MOV R5, A
```

Рисунок 2.3 – Разбитие результата на отдельные байты

Далее, переместив в аккумулятор значение из R7, R6 или R5 (на рисунке 2.4 – пример с R7), оно сравнивается с 0, 1, 2,…, D, E, F с помощью CJNE. При нахождении нужного, значение кода для индикатора нужного числа перемещается в R2, которое затем будет выводиться на индикаторы (рисунок 2.5).

```
77 SETB P0.7
78 MOV A, R7
79 MOV R0, #00h
80 c0:
81     CJNE A, #00h, c1
82     MOV R2, #11000000b
83     JMP ex
84 c1:
85     CJNE A, #01h, c2
86     MOV R2, #11111001b
87     JMP ex
88 c2:
89     CJNE A, #02h, c3
90     MOV R2, #10100100b
91     JMP ex
```

Рисунок 2.4 – Сравнение с шестнадцатеричными значениями

В R0 содержится число, по сути означающее номер байта, который надо вывести. Если он 0, то срабатывается переход на ex и выполнение кода ниже – включение старшего индикатора (P3.3 и P3.4 содержат значение 11 в двоичном виде), вывод на него значения старшего байта, перемещение в аккумулятор значения байта более младшего (R6, а после R5) и переход на c0 (на рисунке

выше). В случае с R6 после перехода на ex будет сравнение R0 с 0, но так как в работе уже второй байт, соответственно, и R0 тоже будет не 0, а 1, поэтому будет переход на ex2, где функционал такой же, как и в ex. Для вывода на P1 значения P3.4 и P3.3 было изменено на 10. Аналогично третьим индикатором – для него значение на P будет 01.

```
ex:
    CJNE R0, #00h, ex2
    SETB P3.3
    SETB P3.4
    MOV P1, R2
    INC R0
    MOV A, R6
    JMP c0
ex2:
    CJNE R0, #01h, ex3
    MOV P1, #0FFH
    CLR P3.3
    MOV P1, R2
    INC R0
    MOV A, R5
    JMP c0
ex3:
    MOV P1, #0FFH
    CLR P3.4
    SETB P3.3
    MOV P1, R2
    MOV P1, #0FFh
```

Рисунок 2.5 – Вывод на индикаторы

Сам вывод представлен на рисунке 2.6.

Рисунок 3.7 – Вывод на индикаторы

Код программы представлен по ссылке в приложении А.

2.4 Заключение

В лабораторной работе №2 был составлен алгоритм программы в соответствии с индивидуальным вариантом (12). Получены навыки работы с памятью через прямые и косвенные адреса, регистрами, логическими и арифметическими операциями над байтами, изучены и реализованы принципы работы с условными и безусловными переходами, а также работа с семисегментными индикаторами.

3 ЛАБОРАТОРНАЯ РАБОТА №3 «ОРГАНИЗАЦИЯ ПОДПРОГРАММ В МИКРОПРОЦЕССОРАХ СЕМЕЙСТВА INTEL MCS-51»

3.1 Цель работы

Изучение принципов организации подпрограмм на примере микропроцессоров семейства Intel mcs-51.

3.2 Задание

Для программы из Лабораторной работы №2 оформите часть кода, отвечающую за определение кода символа, выводимого на семисегментные индикаторы, в виде процедуры.

3.3 Ход работы

В предыдущей лабораторной работе был описан весь основной функционал работы программы. В данной лабораторной будут описаны только изменения в сравнении с прошлой работой, связанные с применением процедур.

После выполнения операций по изменению значений элементов массива, с помощью процедур производилась работа по «подбору» кода для индикаторов в зависимости от значения выводимого шестнадцатиричного числа.

На рисунке 3.1 показано, что по-очередно в аккумулятор загружается значения символа, числа, а потом вызывается процедура PROC0. Из себя она представляет подпрограмму, показанную на рисунке 3.2.

```
67  SETB P0.7
68  MOV R0, #00h
69  MOV R4, #00h
70
71  MOV R1, #40h
72  MOV A, R7
73  CALL PROC0
74
75  MOV R1, #40h
76  MOV A, R6
77  CALL PROC0
78
79  MOV R1, #40h
80  MOV A, R5
81  CALL PROC0
```

Рисунок 3.1 – Занесение в аккумулятор символов и вызов процедуры

```
83  PROC0:
84      ;MOV R7, A
85      CALL PROC1
86      CALL PROC2
87      NOP
88      NOP
89      NOP
90      MOV P1, #0FFh
91      RET
92
93  PROC1:
94      ADD A, R1
95      MOV R0, A
96      MOV A, @R0
97      MOV R2, A
98  RET
```

Рисунок 3.2 – Процедуры PROC0 и PROC1

PROC0 из себя представляет процедуру вызова других процедур и сама по себе смысла не имеет. PROC1 реализует следующий смысл: в R1 находится адрес первого по счету кода для индикатора для выводимого символа. При суммировании, как показано в 94 строчке, с R7, которая была занесена в аккумулятор, адрес будет увеличен на R7. Получившийся адрес будет указывай на ячейку в памяти, в которой будет содержаться необходимый код для индикатора, чтобы вывести как раз символ из R7. По этому адресу затем

вытаскивается этот код и переносится в R2. Затем он считывается и выводится в P1, что из себя представляет индикатор (рисунок 3.3).

```
100 PROC2:
101         CJNE R4, #00h, ex2
102         CLR P3.3
103         SETB P3.4
104         MOV P1, R2
105         INC R4
106         RET
107     ex2:
108         CJNE R4, #01h, ex3
109         SETB P3.3
110         CLR P3.4
111         MOV P1, R2
112         INC R4
113         RET
114     ex3:
115         CJNE R4, #02h, ex4
116         CLR P3.3
117         CLR P3.4
118         MOV P1, R2
119         INC R4
120         RET
```

Рисунок 3.3 – Вывод на индикаторы

Как уже объяснялось в предыдущей лабораторной работе, очистка и наоборот – установка битов в P3.3 и P3.4 отвечает за включение и отключение определенного индикатора. В данной реализации R4 представляет собой счетчик, значение которого после вывода очередного числа увеличивается на 1 и показывает, какой по счету символ будет выводиться следующим.

Стоит также объяснить работу с процедурами. При их вызове в сравнении с переходами (условными или безусловными) разницы никакой – программа начинает выполняться с точки входа, которую указали при вызове с помощью CALL (в данном случае использовадась она, потому что удобно, так как автоматически выбирает одну из двух основных команд ACALL – переход в области одной страницы памяти или LCALL – в области всей программной памяти). По завершении выполнения кода подпрограммы пишется RET, которая смотрит на значение SP – стека. В нем хранится адрес на ячейку в памяти, в которой в свою очередь хранится адрес строчки, к которой будет выполнен переход.

На рисунке 3.4 показан пример во время работы программы до вызова процедуры PROC0. Как видно, значение в SP – по-умолчанию, когда не вызывалась никакая процедура, значение 07 здесь такое, потому что еще ничего не записывалось, он указывает на ячейку памяти, расположенную непосредственно перед стеком. То есть при записи в него значения, будут изменяться 09 и 08 (старший и младший байты).

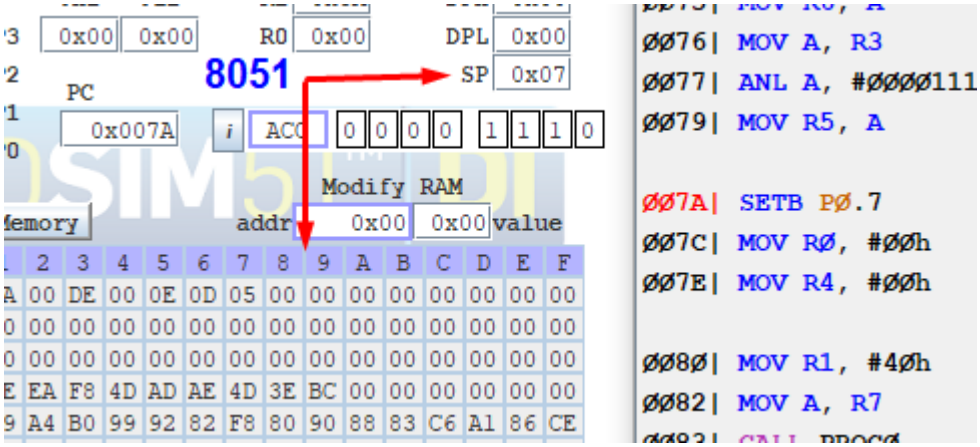

Рисунок 3.4 – До вызова процедуры PROC0

На рисунке 3.5 показано, как изменились значения стека и ячеек после вызова процедуры. Как видно, SP стал равен значению, точнее, адресу, по которому можно пройти в памяти и найти в двух байтах адрес строчки, куда будет выполнен перехо после возвращения из данной подпрограммы, то есть к 0085 (09=00, 08=85). При вызове RET SP снова становится равной 07.

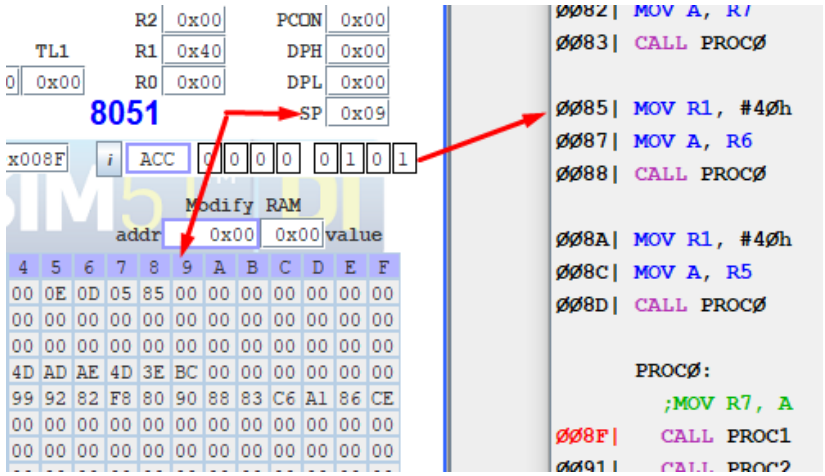

Рисунок 3.5 – После вызова процедуры

При вызове процедуры PROC1, из процедуры PROC0, поменяется значение SP, как видно из рисунка 3.6, на значение, на 2 большее. То есть для

адреса строчки, к которой будет возврат из PROC1 и которая находится в подпрограмме PROC0 выделены другие 2 ячейки памяти, находящиеся по адресу, как указано в SP, 0B. Это и понятно, потому что из PROC0 необходимо также возвращаться и забывать адрес строчки, к которой необходимо возвратиться из нее, нельзя.

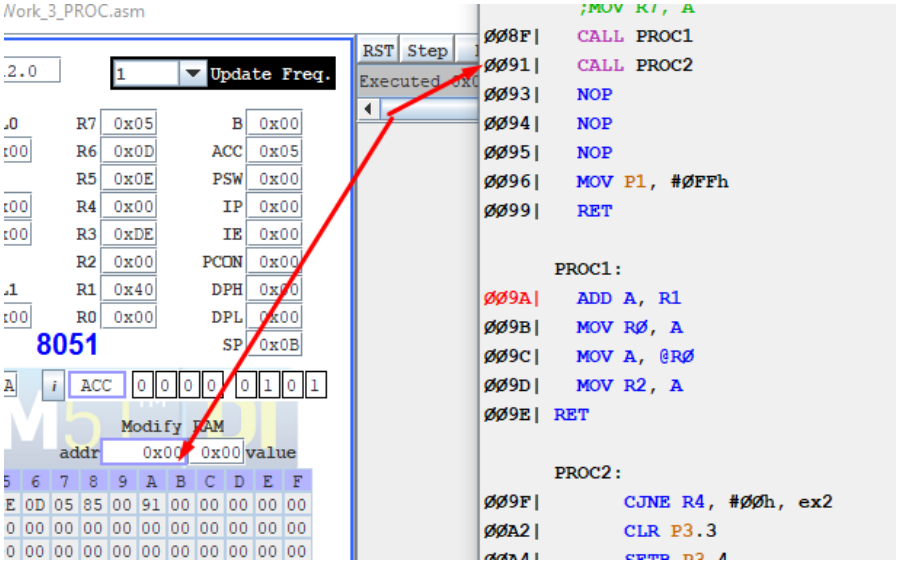

Рисунок 3.6 – При вызове процедуры из процедуры

В результате работы символы были поочередно выведены на индикаторы (рисунок 3.7).

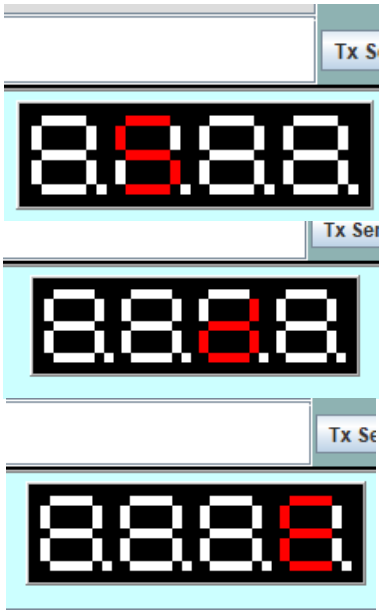

Рисунок 3.7 – Вывод на индикаторы

Код программы представлен по ссылке в приложении А.

3.4 Заключение

В лабораторной работе №3 на основе программы из предыдущей лабораторной работы были изучены принципы работы с процедурами на ассемблере, а также заменена часть кода, отвечающая за вывод символов на семисегментные индикаторы с применением подпрограмм.

ПРИЛОЖЕНИЕ А
(обязательное)

Ссылка на репозиторий с исходным кодом для программ по лабораторным работам

https://github.com/SkripaHella/System-Programming