**Эмулятор процессора**

Вариант 1710 = 100012

* 1 – архитектура фон Неймана
* 00 – стековая архитектура
* 0 – максимальный элимент

**Задание**

Реализовать на языке высокого уровня программу-эмулятор процессора. Написать программу для эмулятора согласно варианту

**Описание регистровой модели процессора с перечнем оборудования (регистров и стеков) и его назначения.**

Схему реализуемого процессора демонстрируется на рисунке 1.

ALU – арифметико-логическое устройство

TOS – top of stack register – содержит верхний элемент стека данных, тогда как указатель DS фактически указывает на второй элемент стека который находится с верху стека

STACK – абстрактный тип данных, представляющий собой список элементов, организованных по принципу LIFO (очередь где первый вошедший элемент выйдет последним). В данной моделе присутствует 2 стека для данных, и для возвратов

MAR – адрес по которому идет предпологается обращаться к памяти

MEMORY – память команд и данных(согласно фон Неймановской архитектуры)

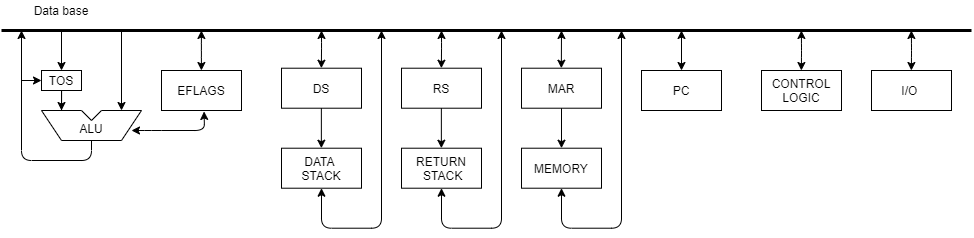


Рис. 1. Блок схема реализуемого процессора

**Список команд**

Таблица 1

|  |  |  |
| --- | --- | --- |
| № | Операция | Описание |
| 0 | ! | Сохранить значение в память по адресу из TOS |
| 1 | @ | Загрузить значение из памяти по адресу из TOS |
| 2 | + | Сложение двух верхних элементов стека |
| 3 | - | Вычетание двух верхних элементов стека |
| 4 | inc | Инкременант значения в регистре TOS |
| 5 | dec | Инкременант значения в регистре TOS |
| 6 | >R | Загрузить в элемент в стек возвратов |
| 7 | R> | Выгрузить в элемент в стек возвратов |
| 8 | AND | Логическое 'и' между двумя верхними элементами стека |
| 9 | OR | Логическое 'или' между двумя верхними элементами стека |
| 10 | XOR | Логическое 'ксор' между двумя верхними элементами стека |
| 11 | DUP | Клонировать верхний элемент стека |
| 12 | OVER | Сделать копию второго элемента стека и положить ее на верх стека |
| 13 | SWAP | Поменять местами два верхних элементов стека |
| 14 | DROP | Выкинуть верхний элемент стека |
| 15 | IFZ | Переход по флагу (Z, S и P) |
| 16 | IFS |
| 17 | IFP |
| 18 | goto | Безусловный переход |
| 19 | CALL | Вызов подпрограммы |
| 20 | EXIT | Выход из подпрограммы |
| 21 | LIT | Загрузить непосредственный операнд |
| 22 | NOP | Конец программы(Пустая строка) |
| 23 | MUL | Умножение |
| 24 | ADC | Сложение |

**Код программы для поиска максимального элемента**

Программа в машинных кодах:

**21, 10, 21, 41, 12, 12, 3, 1, 6, 13, 5, 13, 12, 12, 16, 30, 3, 1, 7, 12, 12, 3, 14, 16, 26, 13, 14, 6, 18, 9, 22, 16, 2, 56, 10, 5, 85, 0, 3, 1, 15**

Массив для поиска максимума: { **16, 2, 56, 10, 5, 85, 0, 3, 1, 15** }

Описание:

1) Загружаем количество элементов массива

2) Загружаем адрес конца массива

3) Копируем верхние 2 элемента в стеке в том же порядке ( a, b, a, b)

4) Получаем адрес первого элемента массива

5) Получаем адрес 1 элемента массива

6) Загружаем элемент массива и закидываем его в стек возвратов

7) Декрементируем счетчик элементов массива

8) Копируем верхние 2 элемента в стеке в том же порядке ( a, b, a, b)

9) Проверяем не дошли ли мы до конца массива

10) Выгружаем из обратного стека число и производим операцию вычетания

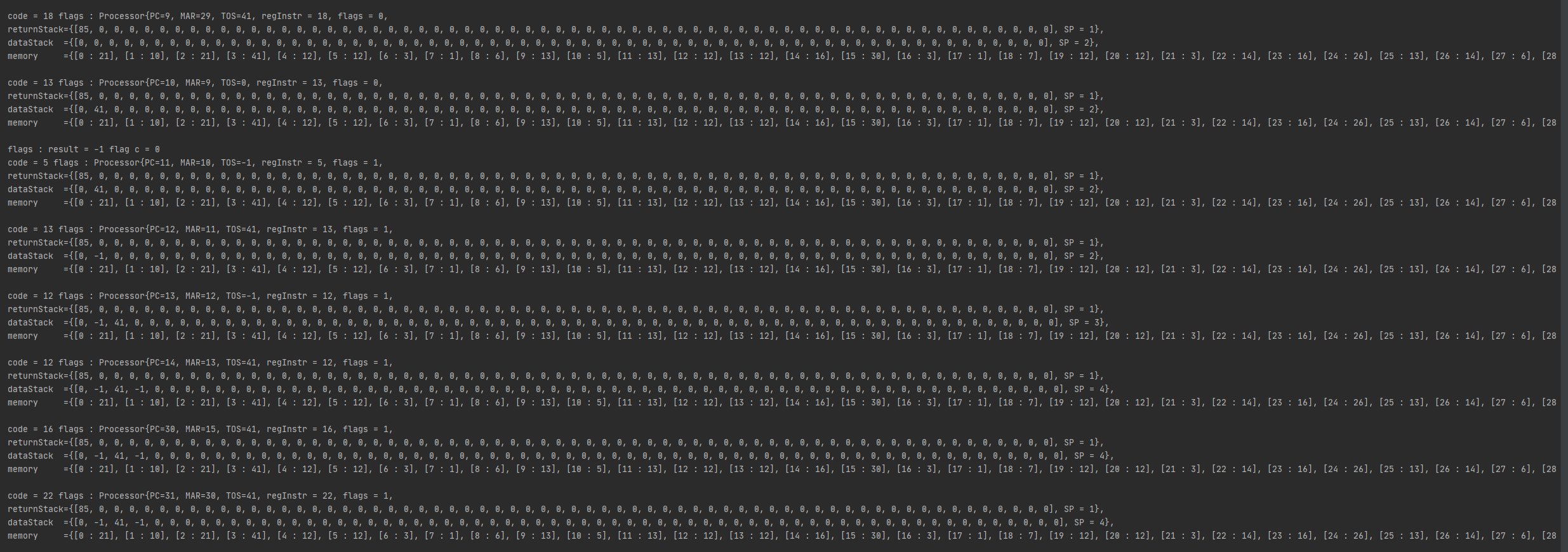
11) В зависимости результата (смотрим флаг ззнака) заносим наибольший элемент в обратный стек

12) Перемещаемся в начало цикла (по адресу 9)

13) Конец программы

14) Массив чисел

**Результат работы программы**

****

**Реализация команд умножения и сложения больших чисел**

**Задание**

Добавить в эмулятор поддержку команд ADC и MUL. Продемонстрировать их работу

**MUL**

Умножение двух 16 битных чисел с получением 32 битного результата.

Старшие 16 бит в регистре TOS (top of stack)

Младшие в верхушке стека

**ADC**

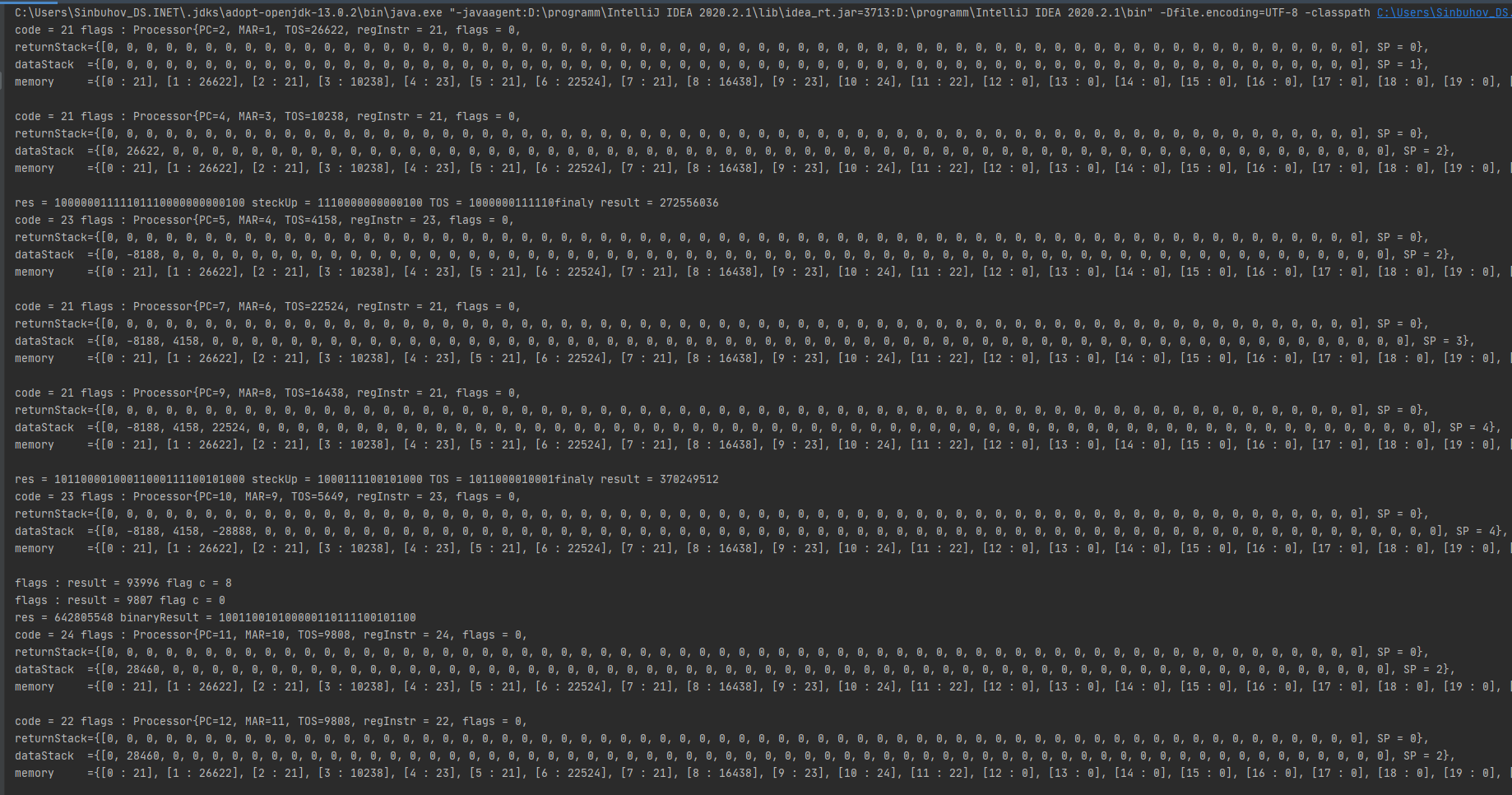
Алгоритм работы:

Сложить R1 : R0 с R3 : R2  
add r2, r0 ; Сложить младший байт с младшим и получить (с установкой флага переполнения)  
add r3, r1 ; Сложить старший байт со старшим и флагом переноса

**Результат работы программы**

Код программы: **21, 26622, 21, 10238, 23, 21, 22524, 21, 16438, 23, 24, 22**

1. Загружаем **2662** и **10238**
2. Перемножаем
3. Загружаем **22524** и **16438**
4. Перемножаем
5. Складываем



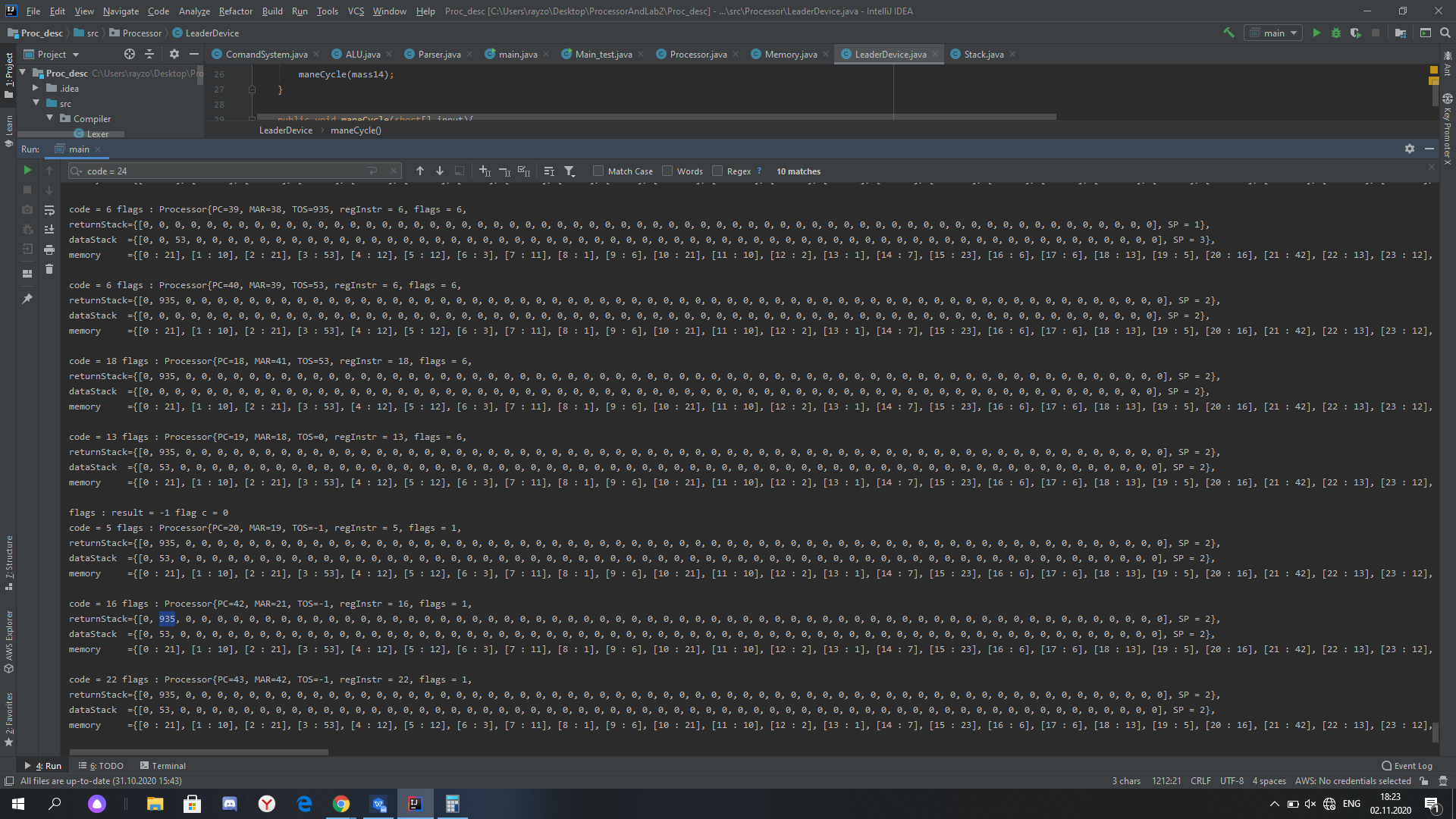
**Выполнение свертки 2 массивов:**

Массив 1: **1, 2, 3, 4, 5, 6, 7, 8, 9, 10**

Массив 2: **11, 12, 13, 14, 15, 16, 17, 18, 19, 20**

**1 \* 11 + 2 \* 12 + … + 10 \* 20 = 935**

**Результат работы программы**



**Простой компилятор для работы с эмулятором**

**Задание**

Реализовать интерпретатор ассеблера для написанного раннее эмулятора

Список токенов:

***MEM\_OP***(Pattern.*compile*(**"^@|!|liter$"**)),  
***ARITHMETIC\_OP***(Pattern.*compile*(**"^add|sub|inc|dec$"**)),  
***BIT\_OP***(Pattern.*compile*(**"or|and|xor"**)),  
***JUMP***(Pattern.*compile*(**"^ifS|ifZ|ifP|goto$"**)),  
***STACK\_OP***(Pattern.*compile*(**"^over|swap|dup|drop$"**)),  
***RETURN\_STACK\_OP***(Pattern.*compile*(**"^pop|push$"**)),  
***EOF***(Pattern.*compile*(**"^nop$"**)),  
***WS***(Pattern.*compile*(**"^\\s+$"**)),***DIGIT***(Pattern.*compile*(**"^0|[1-9][0-9]\*$"**)),  
***CHARS***(Pattern.*compile*(**"^[a-z]+$"**));

Лексер - Выполняет лексический анализ и разбивает текст на токены

Парсер – читает список токенов и переводит программу в машинный код

**Результат работы программы**

**Программа: liter 26622 liter 10238 nop**

**Список токенов:**

**Token: [MEM\_OP , Value : 'liter' ]**

**Token: [WS , Value : ' ' ]**

**Token: [DIGIT , Value : '26622' ]**

**Token: [WS , Value : ' ' ]**

**Token: [MEM\_OP , Value : 'liter' ]**

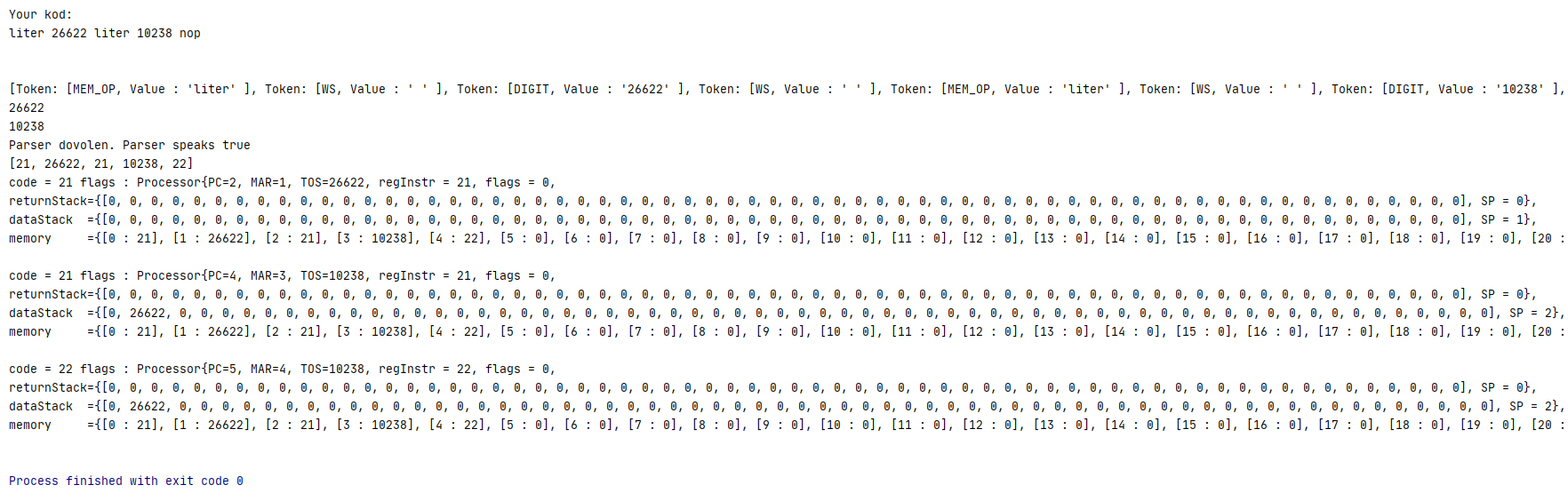
**Token: [WS , Value : ' ' ]**

**Token: [DIGIT , Value : '10238' ]**

**Token: [WS , Value : ' ' ]**

**Token: [EOF , Value : 'nop' ]**

**Перевод в машинный код: [liter** **, 26622** **, liter** **, 10238** **, nop]**

****