

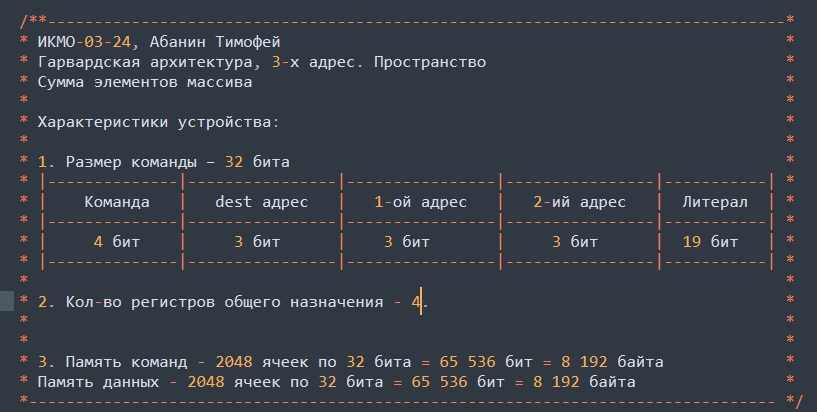
|  |
| --- |
| МИНОБРНАУКИ РОССИИ |
| Федеральное государственное бюджетное образовательное учреждение высшего образования  **«МИРЭА – Российский технологический университет» РТУ МИРЭА** |
| Институт Информационных Технологий |
| Кафедра Вычислительной техники |

|  |  |
| --- | --- |
| **ОТЧЕТ О ВЫПОЛНЕНИИ ПРАКТИЧЕСКОЙ РАБОТЫ** | |
| **по дисциплине** |  |
| **«**Разработка программно-аппаратного обеспечения информационных и автоматизированных систем**»** | |
| Выполнил студент группы ИКМО-03-24 | *Абанин Т.Д.* |
| Принял доцент кафедры ВТ, к.т.н. | *Данилович Е.С.* |

|  |  |
| --- | --- |
| Лабораторная работа выполнена | « » 2024 г. |
| «Зачтено» | « » 2024 г. |

Москва 2024

# ВАРИАНТ ПРАКТИЧЕСКИХ РАБОТ



# НАБОР КОМАНД ЭМУЛЯТОРА

Формат команды: [команда] [операнд1] [операнд2] [операнд2] [значение]:

1. **MOV** – запись значения в операнд
2. **ADD** – запись суммы 2 и 3 операнда в 1 операнд
3. **LOAD** – запись в операнд из памяти
4. **INC** – инкремент операнда
5. **CMP** – сравнение значения в первом и втором операнде
6. **JUMP\_IF\_EQ** – переход к метке, если значения в сравнении равны
7. **JUMP** – переход к метке
8. **STORE** – запись из операнда в память
9. **TERMINATE** – остановка программы

# РЕАЛИЗАЦИЯ ЭМУЛЯТОРА

Листинг 1. Код ассемблера для нахождения суммы массива

MOV R2 0 0

LOAD R3 R2 0 # Загружаем размер массива в регистр R3

INC R2 0 0

LOOP:

CMP R2 R3 0 # Сравниваем значения в 3м и 4м регистрах

JUMP\_IF\_EQ END # Проверяем условие сравнения

LOAD R1 R2 0 # Загружаем значение из текущей ячейки, на которую указывает R2

ADD R0 R0 R1 # Складываем значение из R1 с текущей суммой в R0

INC R2 0 0 # Увеличиваем указатель (R2) на 1

JUMP LOOP # Переход обратно к началу цикла

END:

STORE R0 0 0 # Сохраняем итоговый результат в ячейку 0

TERMINATE # Завершаем выполнение программы

Листинг 2. Машинный код для нахождения суммы массива

10000100000000000000000000000000

00000110100000000000000000000000

01110100000000000000000000000000

01100100110000000000000000000000

00110000000000000000000000001001

00000010100000000000000000000000

00100000000010000000000000000000

01110100000000000000000000000000

01000000000000000000000000000011

00010000000000000000000000000000

01010000000000000000000000000000

Листинг 3. Код компилятора часть 1

*class* Assembler:

*def* \_\_init\_\_(*self*):

*self*.label\_table = {}  *# Таблица меток*

*self*.unresolved\_labels = []  *# Неразрешённые метки*

*self*.reg\_table = {'R0': 0, 'R1': 1, 'R2': 2, 'R3': 3}

*def* compile(*self*, *instructions*):

        machine\_code = []

        command\_index = 0

*# Первый проход: анализ меток и команд*

        for line in instructions:

            trimmed\_line = line.strip()

*#Игнорировать пустые строки и строки с комментариями*

            if not trimmed\_line or trimmed\_line.startswith("#"):

                continue

*# Лексический анализ: распознавание меток*

            if trimmed\_line.endswith(":"):

                label = trimmed\_line[:-1]

                if label in *self*.label\_table:

                    raise *ValueError*(*f*"Метка {label} уже определена.")

*self*.label\_table[label] = command\_index

                continue

            command\_part = trimmed\_line.split("#")[0].strip()

*# Синтаксический анализ команды*

            parts = command\_part.split()

            cmd\_type = parts[0].upper()

            if not *self*.is\_valid\_command(cmd\_type):

                raise *ValueError*(*f*"Неизвестная команда: {cmd\_type}")

*# Обработка операндов с метками*

            operand1 = *self*.parse\_operand(parts[1], command\_index, 0) if len(parts) > 1 else 0

            operand2 = *self*.parse\_operand(parts[2], command\_index, 1) if len(parts) > 2 else 0

            operand3 = *self*.parse\_operand(parts[3], command\_index, 2) if len(parts) > 3 else 0

            binary\_instruction = *self*.generate\_instruction(cmd\_type, operand1, operand2, operand3)

            machine\_code.append(binary\_instruction)

            command\_index += 1

*# Второй проход: разрешение меток*

*self*.resolve\_labels(machine\_code)

        return machine\_code

*def* is\_valid\_command(*self*, *cmd\_type*):

        return cmd\_type in ["LOAD", "STORE", "ADD", "JUMP\_IF\_EQ", "JUMP", "TERMINATE", "CMP", "INC", "MOV"]

*def* generate\_instruction(*self*, *cmd\_type*, *operand1*, *operand2*, *operand3*):

        if cmd\_type == "LOAD":

            binary\_instruction = ((0 << 28) | (operand1 << 25) | (operand2 << 22))

        elif cmd\_type == "STORE":

            binary\_instruction = ((1 << 28) | (operand1 << 25) | (operand2))

        elif cmd\_type == "ADD":

            binary\_instruction = ((2 << 28) | (operand1 << 25) | (operand2 << 22) | (operand3 << 19))

        elif cmd\_type == "JUMP\_IF\_EQ":

            binary\_instruction = ((3 << 28) | (operand1))

        elif cmd\_type == "JUMP":

            binary\_instruction = ((4 << 28) | (operand1))

        elif cmd\_type == "TERMINATE":

            binary\_instruction = (5 << 28)

        elif cmd\_type == "CMP":

            binary\_instruction = ((6 << 28) | (operand1 << 25) | (operand2 << 22))

        elif cmd\_type == "INC":

            binary\_instruction = ((7 << 28) | (operand1 << 25))

        elif cmd\_type == "MOV":

            binary\_instruction = ((8 << 28) | (operand1 << 25) | (operand2))

        else:

            raise *ValueError*(*f*"Неизвестная команда: {cmd\_type}")

        return binary\_instruction

*def* parse\_operand(*self*, *operand*, *command\_index*, *operand\_index*):

*# Если операнд является числом*

        try:

            if operand.isdigit():

                return *int*(operand)

            value = *self*.reg\_table[operand]

            return value

        except *Exception*:

            pass

*# Если операнд является меткой*

        if operand in *self*.label\_table:

            resolved\_address = *self*.label\_table[operand]

            return resolved\_address

*# Если метка ещё не определена, добавляем в нерешённые метки*

*self*.unresolved\_labels.append((operand, command\_index, operand\_index))

        return 0  *# Временно*

*def* resolve\_labels(*self*, *machine\_code*):

        for unresolved in *self*.unresolved\_labels:

            label, command\_index, operand\_index = unresolved

            if label not in *self*.label\_table:

                raise *ValueError*(*f*"Неразрешённая метка: {label}")

            resolved\_address = *self*.label\_table[label]

*# Обновляем нужный операнд*

            instruction = machine\_code[command\_index]

            if operand\_index == 0:  *# Для JUMP*

                machine\_code[command\_index] = (instruction & *0b*11111111111110000000000000000000) | (resolved\_address)

            elif operand\_index == 1:  *# Для JUMP\_IF*

                machine\_code[command\_index] = (instruction & *0b*11111111111110000000000000000000) | (resolved\_address)

class VirtualMachine:

def \_\_init\_\_(self): # Устанавливаем размер памяти по умолчанию

self.pc = 0

self.output\_pc = 0

self.registers = {'A': 0, 'B': 0, 'C': 0}

self.output = [0] \* 2048

self.memory = [0b00] \* 2048 # Инициализация ОЗУ

def execute(self):

while self.pc != len(self.memory):

instruction = self.memory[self.pc]

op = instruction >> 30 # Получаем первые три бита для операции

if op == 0b00: # MOV

reg = self.decode\_register((instruction >> 28) & 0b0011)

value = instruction & 0b00000000111111111111111111111111

self.registers[reg] = value

elif op == 0b01: # ADD

dest\_reg = self.decode\_register((instruction >> 28) & 0b0011)

reg1 = self.decode\_register((instruction >> 26) & 0b000011)

reg2 = self.decode\_register(instruction >> 24 & 0b00000011)

# Сложение значений из reg1 и reg2 и сохранение результата в dest\_reg

self.registers[dest\_reg] = self.registers[reg1] + self.registers[reg2]

elif op == 0b10: # PRINT

reg = self.decode\_register((instruction >> 28) & 0b0011)

self.output[self.output\_pc] = self.registers[reg]

self.output\_pc += 1

self.pc += 1

def get\_output(self):

return self.output

def decode\_register(self, code):

return {

0b00: 'A',

0b01: 'B',

0b10: 'C',

}[code]

def load\_binary\_code\_to\_memory(self, binary\_code):

for i in range(len(binary\_code)):

self.memory[i] = binary\_code[i]

Листинг 4. Код компилятора часть 2

*binary\_instruction = self.generate\_instruction(cmd\_type, operand1, operand2, operand3)*

*machine\_code.append(binary\_instruction)*

*command\_index += 1*

*# Второй проход: разрешение меток*

*self.resolve\_labels(machine\_code)*

*return machine\_code*

*def is\_valid\_command(self, cmd\_type):*

*return cmd\_type in ["LOAD", "STORE", "ADD", "JUMP\_IF\_EQ", "JUMP", "TERMINATE", "CMP", "INC", "MOV"]*

*def generate\_instruction(self, cmd\_type, operand1, operand2, operand3):*

*if cmd\_type == "LOAD":*

*binary\_instruction = ((0 << 28) | (operand1 << 25) | (operand2 << 22))*

*elif cmd\_type == "STORE":*

*binary\_instruction = ((1 << 28) | (operand1 << 25) | (operand2))*

*elif cmd\_type == "ADD":*

*binary\_instruction = ((2 << 28) | (operand1 << 25) | (operand2 << 22) | (operand3 << 19))*

*elif cmd\_type == "JUMP\_IF\_EQ":*

*binary\_instruction = ((3 << 28) | (operand1))*

*elif cmd\_type == "JUMP":*

*binary\_instruction = ((4 << 28) | (operand1))*

*elif cmd\_type == "TERMINATE":*

*binary\_instruction = (5 << 28)*

*elif cmd\_type == "CMP":*

*binary\_instruction = ((6 << 28) | (operand1 << 25) | (operand2 << 22))*

*elif cmd\_type == "INC":*

*binary\_instruction = ((7 << 28) | (operand1 << 25))*

*elif cmd\_type == "MOV":*

*binary\_instruction = ((8 << 28) | (operand1 << 25) | (operand2))*

*else:*

*raise ValueError(f"Неизвестная команда: {cmd\_type}")*

*return binary\_instruction*

*def parse\_operand(self, operand, command\_index, operand\_index):*

*# Если операнд является числом*

*try:*

*if operand.isdigit():*

*return int(operand)*

*value = self.reg\_table[operand]*

*return value*

*except Exception:*

*pass*

*# Если операнд является меткой*

*if operand in self.label\_table:*

*resolved\_address = self.label\_table[operand]*

*return resolved\_address*

*# Если метка ещё не определена, добавляем в нерешённые метки*

*self.unresolved\_labels.append((operand, command\_index, operand\_index))*

*return 0  # Временно*

*def resolve\_labels(self, machine\_code):*

*for unresolved in self.unresolved\_labels:*

*label, command\_index, operand\_index = unresolved*

*if label not in self.label\_table:*

*raise ValueError(f"Неразрешённая метка: {label}")*

*resolved\_address = self.label\_table[label]*

*# Обновляем нужный операнд*

*instruction = machine\_code[command\_index]*

*if operand\_index == 0:  # Для JUMP*

*machine\_code[command\_index] = (instruction & 0b11111111111110000000000000000000) | (resolved\_address)*

*elif operand\_index == 1:  # Для JUMP\_IF*

*machine\_code[command\_index] = (instruction & 0b11111111111110000000000000000000) | (resolved\_address)*

            binary\_instruction = *self*.generate\_instruction(cmd\_type, operand1, operand2, operand3)

            machine\_code.append(binary\_instruction)

            command\_index += 1

*# Второй проход: разрешение меток*

*self*.resolve\_labels(machine\_code)

        return machine\_code

*def* is\_valid\_command(*self*, *cmd\_type*):

        return cmd\_type in ["LOAD", "STORE", "ADD", "JUMP\_IF\_EQ", "JUMP", "TERMINATE", "CMP", "INC", "MOV"]

*def* generate\_instruction(*self*, *cmd\_type*, *operand1*, *operand2*, *operand3*):

        if cmd\_type == "LOAD":

            binary\_instruction = ((0 << 28) | (operand1 << 25) | (operand2 << 22))

        elif cmd\_type == "STORE":

            binary\_instruction = ((1 << 28) | (operand1 << 25) | (operand2))

        elif cmd\_type == "ADD":

            binary\_instruction = ((2 << 28) | (operand1 << 25) | (operand2 << 22) | (operand3 << 19))

        elif cmd\_type == "JUMP\_IF\_EQ":

            binary\_instruction = ((3 << 28) | (operand1))

        elif cmd\_type == "JUMP":

            binary\_instruction = ((4 << 28) | (operand1))

        elif cmd\_type == "TERMINATE":

            binary\_instruction = (5 << 28)

        elif cmd\_type == "CMP":

            binary\_instruction = ((6 << 28) | (operand1 << 25) | (operand2 << 22))

        elif cmd\_type == "INC":

            binary\_instruction = ((7 << 28) | (operand1 << 25))

        elif cmd\_type == "MOV":

            binary\_instruction = ((8 << 28) | (operand1 << 25) | (operand2))

        else:

            raise *ValueError*(*f*"Неизвестная команда: {cmd\_type}")

        return binary\_instruction

*def* parse\_operand(*self*, *operand*, *command\_index*, *operand\_index*):

*# Если операнд является числом*

        try:

            if operand.isdigit():

                return *int*(operand)

            value = *self*.reg\_table[operand]

            return value

        except *Exception*:

            pass

*# Если операнд является меткой*

        if operand in *self*.label\_table:

            resolved\_address = *self*.label\_table[operand]

            return resolved\_address

*# Если метка ещё не определена, добавляем в нерешённые метки*

*self*.unresolved\_labels.append((operand, command\_index, operand\_index))

        return 0  *# Временно*

*def* resolve\_labels(*self*, *machine\_code*):

        for unresolved in *self*.unresolved\_labels:

            label, command\_index, operand\_index = unresolved

            if label not in *self*.label\_table:

                raise *ValueError*(*f*"Неразрешённая метка: {label}")

            resolved\_address = *self*.label\_table[label]

*# Обновляем нужный операнд*

            instruction = machine\_code[command\_index]

            if operand\_index == 0:  *# Для JUMP*

                machine\_code[command\_index] = (instruction & *0b*11111111111110000000000000000000) | (resolved\_address)

            elif operand\_index == 1:  *# Для JUMP\_IF*

                machine\_code[command\_index] = (instruction & *0b*11111111111110000000000000000000) | (resolved\_address)

class VirtualMachine:

def \_\_init\_\_(self): # Устанавливаем размер памяти по умолчанию

self.pc = 0

self.output\_pc = 0

self.registers = {'A': 0, 'B': 0, 'C': 0}

self.output = [0] \* 2048

self.memory = [0b00] \* 2048 # Инициализация ОЗУ

def execute(self):

while self.pc != len(self.memory):

instruction = self.memory[self.pc]

op = instruction >> 30 # Получаем первые три бита для операции

if op == 0b00: # MOV

reg = self.decode\_register((instruction >> 28) & 0b0011)

value = instruction & 0b00000000111111111111111111111111

self.registers[reg] = value

elif op == 0b01: # ADD

dest\_reg = self.decode\_register((instruction >> 28) & 0b0011)

reg1 = self.decode\_register((instruction >> 26) & 0b000011)

reg2 = self.decode\_register(instruction >> 24 & 0b00000011)

# Сложение значений из reg1 и reg2 и сохранение результата в dest\_reg

self.registers[dest\_reg] = self.registers[reg1] + self.registers[reg2]

elif op == 0b10: # PRINT

reg = self.decode\_register((instruction >> 28) & 0b0011)

self.output[self.output\_pc] = self.registers[reg]

self.output\_pc += 1

self.pc += 1

def get\_output(self):

return self.output

def decode\_register(self, code):

return {

0b00: 'A',

0b01: 'B',

0b10: 'C',

}[code]

def load\_binary\_code\_to\_memory(self, binary\_code):

for i in range(len(binary\_code)):

self.memory[i] = binary\_code[i]

Листинг 4. Код компилятора часть 3

*def parse\_operand(self, operand, command\_index, operand\_index):*

*# Если операнд является числом*

*try:*

*if operand.isdigit():*

*return int(operand)*

*value = self.reg\_table[operand]*

*return value*

*except Exception:*

*pass*

*# Если операнд является меткой*

*if operand in self.label\_table:*

*resolved\_address = self.label\_table[operand]*

*return resolved\_address*

*# Если метка ещё не определена, добавляем в нерешённые метки*

*self.unresolved\_labels.append((operand, command\_index, operand\_index))*

*return 0  # Временно*

*def resolve\_labels(self, machine\_code):*

*for unresolved in self.unresolved\_labels:*

*label, command\_index, operand\_index = unresolved*

*if label not in self.label\_table:*

*raise ValueError(f"Неразрешённая метка: {label}")*

*resolved\_address = self.label\_table[label]*

*# Обновляем нужный операнд*

*instruction = machine\_code[command\_index]*

*if operand\_index == 0:  # Для JUMP*

*machine\_code[command\_index] = (instruction & 0b11111111111110000000000000000000) | (resolved\_address)*

*elif operand\_index == 1:  # Для JUMP\_IF*

*machine\_code[command\_index] = (instruction & 0b11111111111110000000000000000000) | (resolved\_address)*

Листинг 5. Код эмулятора часть 1

*class* Processor:

    LOAD = 0      *# 000*

    STORE = 1     *# 001*

    ADD = 2       *# 010*

    JUMP\_IF = 3   *# 011*

    JUMP = 4      *# 100*

    TERMINATE = 5 *# 101*

    CMP = 6       *# 110*

    INC = 7       *# 111*

    MOVE = 8      *# 1000*

*def* \_\_init\_\_(*self*, *RAM\_size* = 2048, *SSD\_size* = 2048):

*self*.reg = {'R0': 0, 'R1': 0, 'R2': 0, 'R3': 0}

*self*.pc = 0  *# Счетчик команд (Program Counter)*

*self*.cmp\_flag = 0  *# Начальное значение флага сравнения*

*self*.data\_memory = [0] \* SSD\_size *# Инициализация массива данных*

*self*.command\_memory = [0] \* RAM\_size *# Инициализация списка команд*

*def* load\_data(*self*, *initial\_data*):

        if len(initial\_data) > len(*self*.data\_memory):

            raise *ValueError*(*f*"Размер загружемых данных больше размера постоянных данных")

*self*.data\_memory[:len(initial\_data)] = initial\_data

*def* load\_program(*self*, *program*):

        if len(program) > len(*self*.command\_memory):

            raise *ValueError*(*f*"Размер программы больше размера оперативной памяти")

        for i in range(len(program)):

*self*.command\_memory[i] = program[i]

*def* fetch\_instruction(*self*, *pc*):

*# Извлечение инструкции по адресу программы*

        binary\_instruction = *self*.command\_memory[pc]

        instruction = []

        instruction.append((binary\_instruction & *0b*11110000000000000000000000000000) >> 28)

        instruction.append((binary\_instruction & *0b*00001110000000000000000000000000) >> 25)

        instruction.append((binary\_instruction & *0b*00000001110000000000000000000000) >> 22)

        instruction.append((binary\_instruction & *0b*00000000001110000000000000000000) >> 19)

        instruction.append((binary\_instruction & *0b*00000000000001111111111111111111))

        return instruction

*def* execute\_program(*self*):

        running = True

        while running:

            instruction = *self*.fetch\_instruction(*self*.pc)  *# Извлечение команды*

            running = *self*.decode\_and\_execute(instruction)  *# Декодирование и выполнение команды*

*self*.pc += 1  *# Переход к следующей команде*

            print()

*def* decode\_register(*self*, *code*):

        return {

            0: 'R0',

            1: 'R1',

            2: 'R2',

            3: 'R3'

        }[code]

*def* decode\_and\_execute(*self*, *instruction*):

        cmd\_type = instruction[0]

        op1 = instruction[1]

        op2 = instruction[2]

        op3 = instruction[3]

        literal = instruction[4]

        if cmd\_type == *self*.LOAD:

*self*.reg[*self*.decode\_register(op1)] = *self*.data\_memory[*self*.reg[*self*.decode\_register(op2)]]  *# Загрузка данных из памяти в регистр*

        elif cmd\_type == *self*.STORE:

*self*.data\_memory[literal] = *self*.reg[*self*.decode\_register(op1)]

        elif cmd\_type == *self*.ADD:

*self*.reg[*self*.decode\_register(op1)] = *self*.reg[*self*.decode\_register(op2)] + *self*.reg[*self*.decode\_register(op3)]

        elif cmd\_type == *self*.MOVE:

*self*.reg[*self*.decode\_register(op1)] = literal

        elif cmd\_type == *self*.TERMINATE:

            return False

        elif cmd\_type == *self*.JUMP:

*self*.pc = literal - 1  *# Переход к указанному адресу*

        elif cmd\_type == *self*.JUMP\_IF:

            if *self*.cmp\_flag == 0:

*self*.pc = literal-1  *# Переход к метке завершения*

                return True *# Не увеличивать PC, т.к. прыжок*

        elif cmd\_type == *self*.INC:

*self*.reg[*self*.decode\_register(op1)] += 1  *# Увеличение значения регистра на 1*

        elif cmd\_type == *self*.CMP:

            if *self*.reg[*self*.decode\_register(op1)] > *self*.reg[*self*.decode\_register(op2)]:

*self*.cmp\_flag = 1  *# Первый регистр больше второго*

            elif *self*.reg[*self*.decode\_register(op1)] < *self*.reg[*self*.decode\_register(op2)]:

*self*.cmp\_flag = -1  *# Первый регистр меньше второго*

            else:

*self*.cmp\_flag = 0  *# Регистр равны*

*# Вывод текущего состояния процессора*

        print(*f*"PC: {*self*.pc}")

        print("Регистры: " + *str*(*self*.reg))

        print("Память данных: " + ", ".join(map(*str*, *self*.data\_memory)))

        return True

Листинг 6. Код эмулятора часть 2

*def* execute\_program(*self*):

        running = True

        while running:

            instruction = *self*.fetch\_instruction(*self*.pc)  *# Извлечение команды*

            running = *self*.decode\_and\_execute(instruction)  *# Декодирование и выполнение команды*

*self*.pc += 1  *# Переход к следующей команде*

            print()

*def* decode\_register(*self*, *code*):

        return {

            0: 'R0',

            1: 'R1',

            2: 'R2',

            3: 'R3'

        }[code]

*def* decode\_and\_execute(*self*, *instruction*):

        cmd\_type = instruction[0]

        op1 = instruction[1]

        op2 = instruction[2]

        op3 = instruction[3]

        literal = instruction[4]

        if cmd\_type == *self*.LOAD:

*self*.reg[*self*.decode\_register(op1)] = *self*.data\_memory[*self*.reg[*self*.decode\_register(op2)]]  *# Загрузка данных из памяти в регистр*

        elif cmd\_type == *self*.STORE:

*self*.data\_memory[literal] = *self*.reg[*self*.decode\_register(op1)]

        elif cmd\_type == *self*.ADD:

*self*.reg[*self*.decode\_register(op1)] = *self*.reg[*self*.decode\_register(op2)] + *self*.reg[*self*.decode\_register(op3)]

        elif cmd\_type == *self*.MOVE:

*self*.reg[*self*.decode\_register(op1)] = literal

        elif cmd\_type == *self*.TERMINATE:

            return False

        elif cmd\_type == *self*.JUMP:

*self*.pc = literal - 1  *# Переход к указанному адресу*

        elif cmd\_type == *self*.JUMP\_IF:

            if *self*.cmp\_flag == 0:

*self*.pc = literal-1  *# Переход к метке завершения*

                return True *# Не увеличивать PC, т.к. прыжок*

        elif cmd\_type == *self*.INC:

*self*.reg[*self*.decode\_register(op1)] += 1  *# Увеличение значения регистра на 1*

        elif cmd\_type == *self*.CMP:

            if *self*.reg[*self*.decode\_register(op1)] > *self*.reg[*self*.decode\_register(op2)]:

*self*.cmp\_flag = 1  *# Первый регистр больше второго*

            elif *self*.reg[*self*.decode\_register(op1)] < *self*.reg[*self*.decode\_register(op2)]:

*self*.cmp\_flag = -1  *# Первый регистр меньше второго*

            else:

*self*.cmp\_flag = 0  *# Регистр равны*

*# Вывод текущего состояния процессора*

        print(*f*"PC: {*self*.pc}")

        print("Регистры: " + *str*(*self*.reg))

        print("Память данных: " + ", ".join(map(*str*, *self*.data\_memory)))

        return True

Листинг 7. Код эмулятора часть 3

 elif cmd\_type == *self*.INC:

*self*.reg[*self*.decode\_register(op1)] += 1  *# Увеличение значения регистра на 1*

        elif cmd\_type == *self*.CMP:

            if *self*.reg[*self*.decode\_register(op1)] > *self*.reg[*self*.decode\_register(op2)]:

*self*.cmp\_flag = 1  *# Первый регистр больше второго*

            elif *self*.reg[*self*.decode\_register(op1)] < *self*.reg[*self*.decode\_register(op2)]:

*self*.cmp\_flag = -1  *# Первый регистр меньше второго*

            else:

*self*.cmp\_flag = 0  *# Регистр равны*

*# Вывод текущего состояния процессора*

        print(*f*"PC: {*self*.pc}")

        print("Регистры: " + *str*(*self*.reg))

        print("Память данных: " + ", ".join(map(*str*, *self*.data\_memory)))

        return True

# ЗАКЛЮЧЕНИЕ

В ходе выполнения практических работ был разработан ассемблер и эмулятор, подходящие под конкретную задачу – нахождение суммы элементов массива.

# Список источников

1. Потехин Д. С., Тарасов И. Е. Разработка программно-аппаратного обеспечения информационных и автоматизированных систем: конспект лекций. – М.: РТУ МИРЭА, 2020. – 136 с.
2. Строгонов А. В. Цифровая обработка сигналов в базисе программируемых логических интегральных схем. – Санкт-Петербург: Лань, 2021. – 312 с.
3. Уэйкерли Джон Ф. Проектирование цифровых устройств. – М.:Постмаркет, 2002. – 533 с.
4. Максфилд К. Проектирование на ПЛИС: курс молодого бойца. – М.: Издательский дом Додэка XXI, 2007. – 408 с.
5. Суворова Е.А. Проектирование цифровых систем на VHDL. – СПб.: БВХ-Петербург, 2003. – 576 с.
6. Бибило П.Н. Основы языка VHDL. – М.: Издательство ЛКИ, 2007. – 224 с.
7. Тарасов И.Е. Разработка цифровых устройств на основе ПЛИС фирмы XILINX с применением языка VHDL. – М.: Горячая линия –Телеком, 2005. – 252 с.
8. Сергиенко А.М. VHDL для проектирования вычислительных устройств. – К.: Корнейчук, ТИД ДС, 2003. – 208 с.
9. Грушвицкий Р.И., Мурсаев А.Х., Угрюмов Е.П. Проектирование систем на микросхемах программируемой структурой. – СПб.: БВХ- Петербург, 2006. – 734 с.
10. Зотов В.Ю. Проектирование встраиваемых микропроцессорных систем на основе ПЛИС фирмы Xilinx в САПР WebPack ISE. – М.: Горячая линияТелеком, 2006. – 520 с.
11. Потехин Д.С., Тарасов И.Е. Разработка систем цифровой обработки сигналов на базе ПЛИС. – М.: Горячая линия-Телеком, 2007. – 248 с.