РОССИЙСКИЙ УНИВЕРСИТЕТ ДРУЖБЫ НАРОДОВ

Факультет физико-математических и естественных наук

Кафедра прикладной информатики и теории вероятностей

ОТЧЕТ
ПО ЛАБОРАТОРНОЙ РАБОТЕ № 8

дисциплина: Архитектура компьютера

Студент: Грицко Сергей

Группа: НКАбд-02-25

МОСКВА

2025 г.

Оглавление

Цель работы

Приобретение навыков написания программ с использованием циклов и обработкой аргументов командной строки.

Теоретическое введение

1.1 Организация Стека

Стек — это фундаментальная структура данных, реализованная на аппаратном уровне в архитектуре процессора, работающая по принципу **LIFO** (Last In — First Out, «последним пришел — первым ушел»). Он необходим для сохранения адресов возврата при вызове процедур, выделения памяти под локальные переменные и временного хранения значений регистров.

Вершина стека — это адрес последнего добавленного элемента, который хранится в регистре **ESP** (Stack Pointer, указатель стека). Дно стека находится на противоположном конце. Основными операциями со стеком являются:

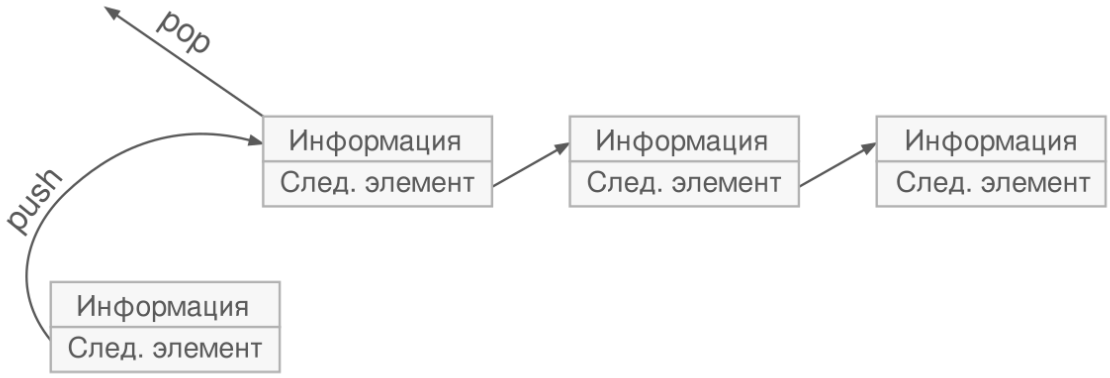

Рисунок 1. Организация стека в процессоре

- **PUSH**: Добавление элемента в вершину стека. При выполнении этой команды указатель стека ESP уменьшается.
- **POP**: Извлечение элемента из вершины стека. При этом указатель стека ESP увеличивается.

Использование стека, как будет показано ниже, критически важно для сохранения контекста регистров при выполнении сложных операций, включая циклы и вызовы функций.

1.2 Программирование циклов

Для организации циклов в ассемблере NASM используется команда **loop**. Эта команда выполняет две основные функции:

1. Уменьшает значение счетчика цикла, хранящегося в регистре **ECX** (Extended Count Register), на 1.
2. Проверяет, равно ли новое значение ECX нулю. Если не равно, происходит переход на метку, указанную в качестве операнда loop. Если равно нулю, выполнение программы продолжается со следующей после loop инструкции.

Поскольку регистр ECX автоматически используется командой loop, его нельзя изменять внутри тела цикла для других целей. Если возникает такая необходимость, требуется временно сохранять (PUSH) и восстанавливать (POP) его значение, чтобы не нарушить логику итераций.

1.3 Обработка аргументов командной строки

При запуске программы из командной строки операционная система использует стек для передачи данных программе. В стек помещаются следующие элементы (снизу вверх):

1. **argc** (Argument Count): Количество аргументов, переданных программе (включая имя самой программы).
2. **argv[0]**: Адрес строки, содержащей имя программы.
3. **argv[1], argv[2], ... argv[n-1]**: Адреса строк, содержащих сами аргументы.

Для доступа к этим данным программа в ассемблере использует инструкцию **pop** для извлечения значений из стека. Сначала извлекается argc (количество), затем argv[0] (имя программы), а затем в цикле — адреса остальных аргументов. Аргументы командной строки всегда передаются как строки, поэтому для выполнения над ними арифметических операций необходимо использовать специальные библиотечные функции (например, atoi из in_out.asm) для преобразования строки в число.

Порядок выполнения лабораторной работы

2.1 Реализация циклов в NASM

Мной была написана программа lab8-1.asm для демонстрации работы циклов и важности сохранения контекста регистра ecx.

При написании программы я столкнулся с тем, что изменение регистра ecx внутри тела цикла (например, при его декременте инструкцией sub ecx, 1) приводило к некорректной работе команды loop, так как она тоже использует ecx как счетчик.

Для обеспечения надежности работы цикла, я использовал стек:

1. Перед изменением: push ecx (сохраняю текущий счетчик).
2. После выполнения операций: pop ecx (восстанавливаю счетчик). Это позволило выполнять внутренние операции с ecx без сбоя логики итераций.

Текст программы lab8-1.asm: *(Смотри файл lab8-1.asm, приложенный к отчету)*

Результат работы программы:

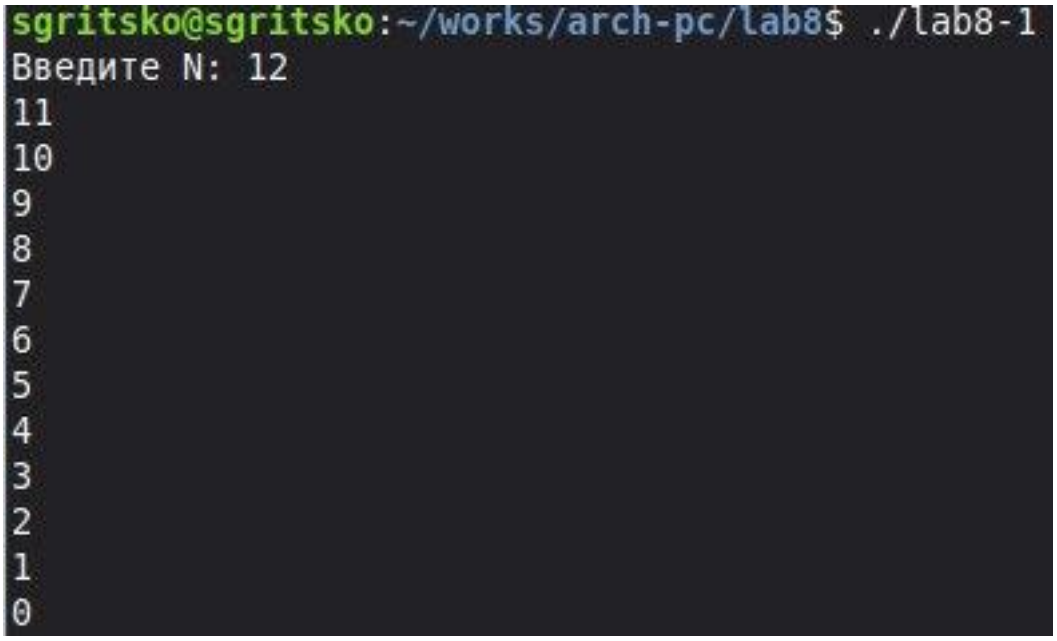

Рисунок 2. Скриншот выполнения программы lab8-1

Вывод: Использование стека позволяет корректно использовать регистр ecx для вычислений внутри цикла, не нарушая работу основного счетчика итераций, что подтверждает успешное выполнение программы.

2.2 Обработка аргументов командной строки

Я написал программу lab8-2.asm, которая демонстрирует, как получить и вывести на экран все аргументы, переданные ей при запуске. Первым делом программа извлекает из стека количество аргументов (argc) и имя самой программы (argv[0]), затем в цикле последовательно извлекает и печатает остальные аргументы.

Текст программы lab8-2.asm: *(Смотри файл lab8-2.asm, приложенный к отчету)*

Результат работы программы:

```
sgritsko@sgritsko:~/works/arch-pc/lab8$ ./lab8-2 12 63 'banana'
12
63
banana
```

Рисунок 2. Скриншот выполнения программы lab8-2

Вывод: Программа корректно обработала и вывела на консоль все переданные ей аргументы, демонстрируя базовый механизм работы с argc и argv через стек.

*2.**3 Арифметические операции с аргументами (Вычисление произведения)***

В данном задании требовалось изменить программу для вычисления суммы аргументов на программу, вычисляющую их **произведение**.

Для этого я выполнил следующие ключевые модификации:

1. Инициализировал регистр результата (esi) значением **1** (нейтральный элемент для умножения).
2. Заменил команду add на команду **imul** (умножение целых чисел).

Текст программы lab8-3.asm: *(Смотри файл lab8-3.asm, приложенный к отчету)*

Результат работы программы:

```
sgritsko@sgritsko:~/works/arch-pc/lab8$ ./lab8-3 4 1 5 10
Результат (произведение): 200
```

Рисунок 3. Скриншот выполнения программы lab8-3

Вывод: Путем изменения инициализации и основной арифметической инструкции программа была успешно модифицирована для вычисления произведения аргументов, что подтверждено результатом.

Порядок выполнения самостоятельной работы

Вариант №14

Задание: Написать программу, которая находит сумму значений функции f(x) для переданных аргументов x_1, x_2, \dots, x_n. **Вариант 14:** Функция f(x) = x + 12.

Алгоритм решения:

1. Инициализировать регистр суммы (esi) нулем.
2. Организовать цикл по всем аргументам командной строки.
3. В цикле каждый аргумент x преобразовать из строки в число.
4. Вычислить f(x) = x + 12 (простая операция сложения).
5. Добавить полученное значение f(x) к регистру суммы esi.
6. Вывести финальную сумму.

Текст программы lab8-4.asm: *(Смотри файл lab8-4.asm, приложенный к отчету)*

Результат работы программы:

```
sgritsko@sgritsko:~/works/arch-pc/lab8$ ./lab8-4 1 2
Функция: f(x) = x + 12
Результат (сумма): 27
```

Рисунок 4. Скриншот выполнения программы lab8-4

Результат выполнения: Для проверки использовались аргументы x=1 и x=2. f(1) = 1 + 12 = 13 f(2) = 2 + 12 = 14 Общая сумма: 13 + 14 = 27. Команда для проверки: ./lab8-4 1 2

Вывод: Программа корректно обрабатывает аргументы командной строки, преобразует их в формат и вычисляет сумму значений функции f(x) = x + 12 для всех переданных аргументов.

Выводы

В ходе выполнения лабораторной работы я успешно реализовал и отладил четыре программы на языке ассемблера NASM. Были освоены следующие ключевые навыки и концепции:

1. **Программирование циклов** с использованием команды loop и корректное управление регистром-счетчиком ecx посредством стека (push/pop).
2. **Обработка аргументов командной строки** через стек, включая извлечение количества аргументов (argc) и самих аргументов (argv).
3. **Выполнение арифметических операций** над числовыми аргументами командной строки после их преобразования из строкового формата. Все поставленные задачи выполнены, цель работы достигнута.

Список литературы

1. GDB: The GNU Project Debugger. — URL: https://www.gnu.org/software/gdb/.
2. NASM Assembly Language Tutorials. — 2021. — URL: https://asmtutor.com/.
3. Колдаев В. Д., Лупин С. А. Архитектура ЭВМ. — М. : Форум, 2018.
4. Куляс О. Л., Никитин К. А. Курс программирования на ASSEMBLER. — М. : Солон- Пресс, 2017.
5. Столяров А. Программирование на языке ассемблера NASM для ОС Unix. — 2-е изд. М. : МАКС Пресс, 2011.
6. Таненбаум Э. Архитектура компьютера. — 6-е изд. — СПб. : Питер, 2013. — 874 с. 12