

Отчёт по лабораторной работе №4

Создание и процесс обработки программ на языке ассемблера NASM

Саакян Нерсес Варданович

Содержание

1	Цель работы	5
2	Задание	6
3	Теоретическое введение	7
4	Выполнение лабораторной работы	13
5	Выводы	18
	Список литературы	19

Список иллюстраций

4.1	Создание каталога	13
4.2	Переходим в каталог	13
4.3	Создание текстового файла	13
4.4	Открытие файла	13
4.5	Ввод текста	14
4.6	Компиляция текста	14
4.7	Проверка, что файл был создан	14
4.8	Создание файла	14
4.9	Передача файла на компоновку	15
4.10	Проверка, что исполняемый файл hello был создан	15
4.11	Зададим имя создаваемого исполняемого файл	15
4.12	Запуск на выполнение созданный исполняемый файл	15
4.13	Создание копии файла с именем lab4.asm	15
4.14	Внесение изменений в текст программы	16
4.15	Оттранслирование, компоновка, запуск	16
4.16	Копирование файлов в локальный репозиторий	16
4.17	Загрузка файлов на Github	17

Список таблиц

1 Цель работы

Освоение процедуры компиляции и сборки программ, написанных на ассемблере NASM.

2 Задание

Программа Hello world!

Транслятор NASM

Расширенный синтаксис командной строки NASM

Компоновщик LD

Запуск исполняемого файла

3 Теоретическое введение

• Основные принципы работы компьютера Основными функциональными элементами любой электронно-вычислительной машины (ЭВМ) являются центральный процессор, память и периферийные устройства (рис. 4.1). Взаимодействие этих устройств осуществляется через общую шину, к которой они подключены. Физически шина представляет собой большое количество проводников, соединяющих устройства друг с другом. В современных компьютерах проводники выполнены в виде электропроводящих дорожек на материнской (системной) плате. Основной задачей процессора является обработка информации, а также организация координации всех узлов компьютера. В состав центрального процессора (ЦП) входят следующие устройства:

- арифметико-логическое устройство (АЛУ) — выполняет логические и арифметические действия, необходимые для обработки информации, хранящейся в памяти;
- устройство управления (УУ) — обеспечивает управление и контроль всех устройств компьютера;
- регистры — сверхбыстрая оперативная память небольшого объёма, входящая в состав процессора, для временного хранения промежуточных результатов выполнения инструкций; регистры процессора делятся на два типа: регистры общего назначения и специальные регистры. Для того, чтобы писать программы на ассемблере, необходимо знать, какие регистры

процессора существуют и как их можно использовать. Большинство команд в программах написанных на ассемблере используют регистры в качестве операндов. Практически все команды представляют собой преобразование данных хранящихся в регистрах процессора, это например пересылка данных между ре-

гистрами или между регистрами и памятью, пре- образование (арифметические или логические операции) данных хранящихся в регистрах. Доступ к регистрам осуществляется не по адресам, как к основной памяти, а по именам.

Каждый регистр процессора архитектуры x86 имеет свое название, состоящее из 2 или 3 букв латинского алфавита. В качестве примера приведем названия основных регистров общего назначения (именно эти регистры чаще всего используются при написании программ): • RAX, RCX, RDX, RBX, RSI, RDI — 64-битные • EAX, ECX, EDX, EBX, ESI, EDI — 32-битные • AX, CX, DX, BX, SI, DI — 16-битные • AH, AL, CH, CL, DH, DL, BH, BL — 8-битные (половинки 16-битных регистров). Например, AH (high AX) — старшие 8 бит регистра AX, AL (low AX) — младшие 8 бит регистра AX. Таким образом можно отметить, что вы можете написать в своей программе, например,

такие команды (mov – команда пересылки данных на языке ассемблера): mov ax, 1 mov eax, 1 Обе команды поместят в регистр AX число 1. Разница будет заключаться только в том, что вторая команда обнулит старшие разряды регистра EAX, то есть после выполнения второй команды в регистре EAX будет число 1. А первая команда оставит в старших разрядах регистра EAX старые данные. И если там были данные, отличные от нуля, то после выполнения первой команды в регистре EAX будет какое-то число, но не 1. А вот в регистре AX будет число 1. Другим важным узлом ЭВМ является оперативное запоминающее устройство (ОЗУ). ОЗУ — это быстродействующее энергозависимое запоминающее устройство, которое напрямую взаимодействует с узлами процессора, предназначенное для хранения программ и данных, с которыми процессор непосредственно работает в текущий момент. ОЗУ состоит из одинаковых пронумерованных ячеек памяти. Номер ячейки памяти — это адрес хранящихся в ней данных. В состав ЭВМ также входят периферийные устройства, которые можно разделить на: • устройства внешней памяти, которые предназначены для долговременного хранения больших объемов данных (жёсткие диски, твердотельные накопители, магнитные ленты); • устройства ввода-вывода, которые обеспечивают взаимодействие ЦП с внешней

средой. В основе вычислительного процесса ЭВМ лежит принцип программного управления. Это означает, что компьютер решает поставленную задачу как последовательность действий, записанных в виде программы. Программа состоит из машинных команд, которые указывают, какие операции и над какими данными (или операндами), в какой последовательности необходимо выполнить. Набор машинных команд определяется устройством конкретного процессора. Коды команд представляют собой многоразрядные двоичные комбинации из 0 и 1. В коде машинной команды можно выделить две части: операционную и адресную. В операционной части хранится код команды, которую необходимо выполнить. В адресной части хранятся данные или адреса данных, которые участвуют в выполнении данной операции. При выполнении каждой команды процессор выполняет определённую последовательность стандартных действий, которая называется командным циклом процессора. В самом общем виде он заключается в следующем: 1. формирование адреса в памяти очередной команды; 2. считывание кода команды из памяти и её дешифрация; 3. выполнение команды; 4. переход к следующей команде.

Данный алгоритм позволяет выполнить хранящуюся в ОЗУ программу. Кроме того, в зависимости от команды при её выполнении могут проходить не все этапы. Более подробно введение о теоретических основах архитектуры ЭВМ см. в [9; 11]. • 4.2.2. Ассемблер и язык ассемблера Язык ассемблера (assembly language, сокращённо asm) — машинно-ориентированный язык низкого уровня. Можно считать, что он больше любых других языков приближен к архитектуре ЭВМ и её аппаратным возможностям, что позволяет получить к ним более полный доступ, нежели в языках высокого уровня, таких как C/C++, Perl, Python и пр. Заметим, что получить полный доступ к ресурсам компьютера в современных архитектурах нельзя, самым низким уровнем работы прикладной программы является обращение напрямую к ядру операционной системы. Именно на этом уровне и работают программы, написанные на ассемблере. Но в отличие от языков высокого уровня ассемблерная программа содержит только тот код, который ввёл программист.

Таким образом язык ассемблера — это язык, с помощью которого понятным для человека образом пишутся команды для процессора. Следует отметить, что процессор понимает не команды ассемблера, а последовательности из нулей и единиц — машинные коды. До появления языков ассемблера программистам приходилось писать программы, используя только лишь машинные коды, которые были крайне сложны для запоминания, так как представляли собой числа, записанные в двоичной или шестнадцатеричной системе счисления. Преобразование или трансляция команд с языка ассемблера в исполняемый машинный код осуществляется специальной программой транслятором — Ассемблер. Программы, написанные на языке ассемблера, не уступают в качестве и скорости программам, написанным на машинном языке, так как транслятор просто переводит мнемонические обозначения команд в последовательности бит (нулей и единиц). Используемые мнемоники обычно одинаковы для всех процессоров одной архитектуры или семейства архитектур (среди широко известных — мнемоники процессоров и контроллеров x86, ARM, SPARC, PowerPC, M68k). Таким образом для каждой архитектуры существует свой ассемблер и, соответственно, свой язык ассемблера. Наиболее распространёнными ассемблерами для архитектуры x86 являются: • для DOS/Windows: Borland Turbo Assembler (TASM), Microsoft Macro Assembler (MASM) и Watcom assembler (WASM); • для GNU/Linux: gas (GNU Assembler), использующий AT&T-синтаксис, в отличие от большинства других популярных ассемблеров, которые используют Intel-синтаксис. Более подробно о языке ассемблера см., например, в [10]. В нашем курсе будет использоваться ассемблер NASM (Netwide Assembler) [7; 12; 14]. NASM — это открытый проект ассемблера, версии которого доступны под различные операционные системы и который позволяет получать объектные файлы для этих систем. В NASM используется Intel-синтаксис и поддерживаются инструкции x86-64. Типичный формат записи команд NASM имеет вид:

Здесь мнемокод — непосредственно мнемоника инструкции процессору, которая является

обязательной частью команды. Операндами могут быть числа, данные, адреса регистров или адреса оперативной памяти. Метка — это идентификатор, с которым ассемблер ассоциирует некоторое число, чаще всего адрес в памяти. Т.о. метка перед командой связана с адресом данной команды. Допустимыми символами в метках являются буквы, цифры, а также следующие символы: *, You can't use 'macro parameter character #' in math mode \$, #, @, ~, . и ?*. Начинаться метка или идентификатор могут с буквы, *.*, и *?*. Перед идентификаторами, которые пишутся как зарезервированные слова, нужно писать *\$*, чтобы компилятор трактовал его верно (так называемое экранирование). Максимальная длина идентификатора 4095 символов. Программа на языке ассемблера также может содержать директивы — инструкции, не переводящиеся непосредственно в машинные команды, а управляющие работой транслятора. Например, директивы используются для определения данных (констант и переменных) и обычно пишутся большими буквами.

• 4.2.3. Процесс создания и обработки программы на языке ассемблера

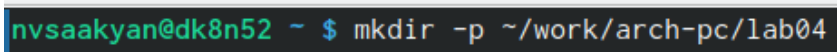
В процессе создания ассемблерной программы можно выделить четыре шага:

- Набор текста программы в текстовом редакторе и сохранение её в отдельном файле. Каждый файл имеет свой тип (или расширение), который определяет назначение файла. Файлы с исходным текстом программ на языке ассемблера имеют тип *asm*.
- Трансляция — преобразование с помощью транслятора, например *nasm*, текста программы в машинный код, называемый объектным. На данном этапе также может быть получен листинг программы, содержащий кроме текста программы различную дополнительную информацию, созданную транслятором. Тип объектного файла — *o*, файла листинга — *lst*.
- Компоновка или линковка — этап обработки объектного кода компоновщиком (*ld*), который принимает на вход объектные файлы и собирает по ним исполняемый файл. Исполняемый файл обычно не имеет расширения. Кроме того, можно получить файл карты загрузки программы в ОЗУ, имеющий расширение *map*.
- Запуск программы. Конечной целью является работоспособный исполняемый файл. Ошибки на предыдущих этапах могут привести к некорректной работе программы, поэтому может при-

существовать этап отладки программы при помощи специальной программы — отладчика. При нахождении ошибки необходимо провести коррекцию программы, начиная с первого шага. Из-за специфики программирования, а также по традиции для создания программ на языке ассемблера обычно пользуются утилитами командной строки (хотя поддержка ассемблера есть в некоторых универсальных интегрированных средах).

4 Выполнение лабораторной работы

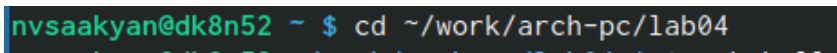
Создайте каталог для работы спrogramмами на языке ассемблера NASM: (рис. 4.1).



```
nvsaakyan@dk8n52 ~ $ mkdir -p ~/work/arch-pc/lab04
```

Рис. 4.1: Создание каталога

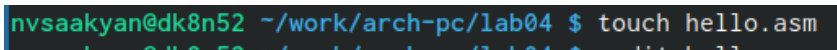
Перейдем в созданный каталог: (рис. 4.2).



```
nvsaakyan@dk8n52 ~ $ cd ~/work/arch-pc/lab04
```

Рис. 4.2: Переходим в каталог

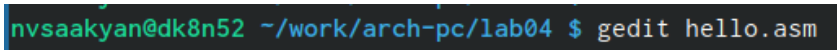
Создадим текстовый файл с именем hello.asm: (рис. 4.3).



```
nvsaakyan@dk8n52 ~/work/arch-pc/lab04 $ touch hello.asm
```

Рис. 4.3: Создание текстового файла

Откроем этот файл с помощью текстового редактора: (рис. 4.4).



```
nvsaakyan@dk8n52 ~/work/arch-pc/lab04 $ gedit hello.asm
```

Рис. 4.4: Открытие файла

Введем в него текст: (рис. 4.5).

```

1 hello.asm
2 SECTION .data ; Начало секции данных
3 hello: DB 'Hello world!',10 ; 'Hello world!' плюс
4 ; символ перевода строки
5 helloLen: EQU $-hello ; Длина строки hello
6 SECTION .text ; Начало секции кода
7 GLOBAL _start
8 _start: ; Точка входа в программу
9 mov eax,4 ; Системный вызов для записи (sys_write)
10 mov ebx,1 ; Описатель файла '1' - стандартный вывод
11 mov ecx,hello ; Адрес строки hello в ecx
12 mov edx,helloLen ; Размер строки hello
13 int 80h ; Вызов ядра
14 mov eax,1 ; Системный вызов для выхода (sys_exit)
15 mov ebx,0 ; Выход с кодом возврата '0' (без ошибок)
16 int 80h ; Вызов ядра

```

Рис. 4.5: Ввод текста

Скомпилируем данный текст: (рис. 4.6).

```

nvsaakyan@dk8n52 ~/work/arch-pc/lab04 $ nasm -f elf hello.asm

```

Рис. 4.6: Компиляция текста

Проверим, что объектный файл был создан: (рис. 4.7).

```

nvsaakyan@dk8n52 ~/work/arch-pc/lab04 $ ls
hello.asm hello.o

```

Рис. 4.7: Проверка, что файл был создан

Скомпилируем исходный файл hello.asm в obj.o и создадим файл листинга list.lst: (рис. 4.8).

```

nvsaakyan@dk8n52 ~/work/arch-pc/lab04 $ nasm -o obj.o -f elf -g -l list.lst hello.asm

```

Рис. 4.8: Создание файла

Проверим, что файл был создан: (рис. ??).

[Проверка, что файлы были созданы]](image/9.png){#fig:009 width=70%}

Передадим объектный файл на обработку компоновщику: (рис. 4.9).

```
nvsaakyan@dk8n52 ~/work/arch-pc/lab04 $ ld -m elf_i386 hello.o -o hello
```

Рис. 4.9: Передача файла на компоновку

Проверим, что исполняемый файл hello был создан: (рис. 4.10).

```
nvsaakyan@dk8n52 ~/work/arch-pc/lab04 $ ls
hello  hello.asm  hello.o  list.lst  obj.o
```

Рис. 4.10: Проверка, что исполняемый файл hello был создан

Зададим имя создаваемого исполняемого файл: (рис. 4.11).

```
nvsaakyan@dk8n52 ~/work/arch-pc/lab04 $ ld -m elf_i386 obj.o -o main
```

Рис. 4.11: Зададим имя создаваемого исполняемого файл

Запустим на выполнение созданный исполняемый файл, находящийся в текущем каталоге: (рис. 4.12).

```
nvsaakyan@dk8n52 ~/work/arch-pc/lab04 $ ./hello
Hello world!
```

Рис. 4.12: Запуск на выполнение созданный исполняемый файл

Создадим копию файла hello.asm с именем lab4.asm: (рис. 4.13).

```
nvsaakyan@dk8n52 ~/work/arch-pc/lab04 $ cp hello.asm lab4.asm
```

Рис. 4.13: Создание копии файла с именем lab4.asm

Внесем изменения в текст программы в файл lab4.asm: (рис. 4.14).

```

1 hello.asm
2 SECTION .data ; Начало секции данных
3 hello: DB 'Saakyan',10 ; 'Hello world!' плюс
4 ; символ перевода строки
5 helloLen: EQU $-hello ; Длина строки hello
6 SECTION .text ; Начало секции кода
7 GLOBAL _start
8 _start: ; Точка входа в программу
9 mov eax,4 ; Системный вызов для записи (sys_write)
10 mov ebx,1 ; Описатель файла '1' - стандартный вывод
11 mov ecx,hello ; Адрес строки hello в ecx
12 mov edx,helloLen ; Размер строки hello
13 int 80h ; Вызов ядра
14 mov eax,1 ; Системный вызов для выхода (sys_exit)
15 mov ebx,0 ; Выход с кодом возврата '0' (без ошибок)
16 int 80h ; Вызов ядра

```

Рис. 4.14: Внесение изменений в текст программы

Оттранслируем полученный текст программы lab4.asm в объектный файл. Выполним компоновку объектного файла и запустим получившийся исполняемый файл: (рис. 4.15).

```

nvsaakyan@dk8n52 ~/work/arch-pc/lab04 $ gedit hello.asm
nvsaakyan@dk8n52 ~/work/arch-pc/lab04 $ cp hello.asm lab4.asm
nvsaakyan@dk8n52 ~/work/arch-pc/lab04 $ nasm -f elf lab4.asm
lab4.asm:1: warning: label alone on a line without a colon might be in error [-w+label-orphan]
nvsaakyan@dk8n52 ~/work/arch-pc/lab04 $ nasm -o obj.o -f elf -g -l list.lst lab4.asm
lab4.asm:1: warning: label alone on a line without a colon might be in error [-w+label-orphan]
nvsaakyan@dk8n52 ~/work/arch-pc/lab04 $ nasm -o Saakyan.o -f elf -g -l list.lst lab4.asm
lab4.asm:1: warning: label alone on a line without a colon might be in error [-w+label-orphan]
nvsaakyan@dk8n52 ~/work/arch-pc/lab04 $ ld -m elf_i386 Saakyan.o -o Saakyan
nvsaakyan@dk8n52 ~/work/arch-pc/lab04 $ ./Saakyan

```

Рис. 4.15: Оттранслирование, компоновка, запуск

Скопируем файлы hello.asm и lab4.asm в локальный репозиторий в каталог ~/work/study/2023-2024/“Архитектура компьютера”/arch-pc/labs/lab04/ с помощью утилиты cp и проверим наличие файлов с помощью утилиты ls: (рис. 4.16).

```

nvsaakyan@dk8n52 ~/work/arch-pc/lab04 $ cp hello.asm ~/work/study/2023-2024/“Архитектура компьютера”/arch-pc/labs/lab04/report
nvsaakyan@dk8n52 ~/work/arch-pc/lab04 $ cp lab4.asm ~/work/study/2023-2024/“Архитектура компьютера”/arch-pc/labs/lab04/report
nvsaakyan@dk8n52 ~/work/arch-pc/lab04 $ cd ~/work/study/2023-2024/“Архитектура компьютера”/arch-pc/labs/lab04/report
nvsaakyan@dk8n52 ~/work/study/2023-2024/Архитектура компьютера/arch-pc/labs/lab04/report $ ls

```

Рис. 4.16: Копирование файлов в локальный репозиторий

Загрузим файлы на Github: (рис. 4.17).


```
nvsaaayan@dk8n52 ~/work/study/2023-2024/Архитектура компьютера/arch-pc/labs/lab04/report $ git add .
nvsaaayan@dk8n52 ~/work/study/2023-2024/Архитектура компьютера/arch-pc/labs/lab04/report $ git commit -am 'feat(main): make course struc
[master 8f7f46c] feat(main): make course structure
5 files changed, 32 insertions(+), 133 deletions(-)
delete mode 100644 labs/lab02/report/report.docx
delete mode 100644 labs/lab02/report/report.md
delete mode 100644 labs/lab02/report/report.pdf
create mode 100644 labs/lab04/report/hello.asm
create mode 100644 labs/lab04/report/lab4.asm
nvsaaayan@dk8n52 ~/work/study/2023-2024/Архитектура компьютера/arch-pc/labs/lab04/report $ git push
git push origin main:main
```

Рис. 4.17: Загрузка файлов на Github

5 Выводы

В ходе выполнения работы, я освоил процедуры компиляции и сборки программ, написанных на ассемблере NASM.

Список литературы