5. Лабораторная работа №5. Создание и процесс обработки программ на языке ассемблера NASM

5.1. Цель работы

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

5.2. Теоретическое введение

5.2.1. Основные принципы работы компьютера

Основными функциональными элементами любой электронно-вычислительной машины (ЭВМ) являются центральный процессор, память и периферийные устройства (рис. 5.1).

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.

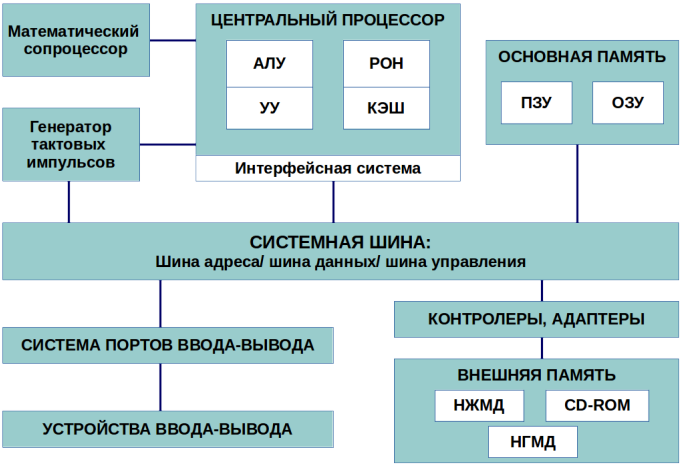

Рис. 5.1. Структурная схема ЭВМ

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 регистрами или между регистрами и памятью, преобразование (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.

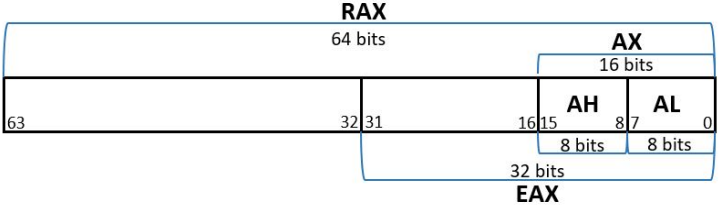

Рис. 5.2. 64-битный регистр процессора ‘RAX’

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 объёмов 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].

5.2.2. 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:

```
[метка:]  мнемокод [операнд {, операнд}]  [; комментарий]
```

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: _, $, #, @,~, . 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.

5.2.3. Процесс создания и обработки программы на языке ассемблера

Процесс создания ассемблерной программы можно изобразить в виде следующей схемы (рис. 5.3).

Рис. 5.3. Процесс создания ассемблерной программы

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 присутствовать 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).

5.3. Порядок выполнения лабораторной работы

5.3.1. Программа Hello world!

Рассмотрим пример простой программы на языке ассемблера NASM. Традиционно первая программа выводит приветственное сообщение `Hello world!` на экран.

Создайте каталог для работы с программами на языке ассемблера NASM:

```
mkdir  ~/work/arch-pc/lab05
```

Перейдите в созданный каталог

```
cd  ~/work/arch-pc/lab05
```

Создайте текстовый файл с именем `hello.asm`

```
touch hello.asm
```

откройте этот файл с помощью любого текстового редактора, например, gedit

```
gedit hello.asm
```

и введите в него следующий текст:

```
; hello.asm
SECTION .data                       ; Начало секции данных
    hello:     DB 'Hello world!',10 ; 'Hello world!' плюс
                                    ; символ перевода строки
    helloLen:  EQU $-hello          ; Длина строки hello

SECTION .text           ; Начало секции кода
    GLOBAL _start

_start:                 ; Точка входа в программу
    mov eax,4           ; Системный вызов для записи (sys_write)
    mov ebx,1           ; Описатель файла '1' - стандартный вывод
    mov ecx,hello       ; Адрес строки hello в ecx
    mov edx,helloLen    ; Размер строки hello
    int 80h             ; Вызов ядра

    mov eax,1           ; Системный вызов для выхода (sys_exit)
    mov ebx,0           ; Выход с кодом возврата '0' (без ошибок)
    int 80h             ; Вызов ядра
```

В отличие от многих современных высокоуровневых языков программирования, в ассемблерной программе каждая команда располагается на *отдельной строке*. Размещение нескольких команд на одной строке **недопустимо**. Синтаксис ассемблера NASM является *чувствительным к регистру*, т.е. есть разница между большими и малыми буквами.

5.3.2. Транслятор NASM

NASM превращает текст программы в объектный код. Например, для компиляции приведённого выше текста программы «Hello World» необходимо написать:

```
nasm -f elf hello.asm
```

Если текст программы набран без ошибок, то транслятор преобразует текст программы из файла `hello.asm` в объектный код, который запишется в файл `hello.o`. Таким образом, имена всех файлов получаются из имени входного файла и расширения по умолчанию. При наличии ошибок объектный файл не создаётся, а после запуска транслятора появятся сообщения об ошибках или предупреждения.

С помощью команды `ls` проверьте, что объектный файл был создан. Какое имя имеет объектный файл?

NASM не запускают без параметров. Ключ `-f` указывает транслятору, что требуется создать бинарные файлы в формате ELF. Следует отметить, что формат `elf64` позволяет создавать исполняемый код, работающий под 64-битными версиями Linux. Для 32-битных версий ОС указываем в качестве формата просто `elf`.

NASM всегда создаёт выходные файлы в **текущем** каталоге.

5.3.3. Расширенный синтаксис командной строки NASM

Полный вариант командной строки nasm выглядит следующим образом:

```
nasm [-@ косвенный_файл_настроек] [-o объектный_файл] [-f
 ↪ формат_объектного_файла] [-l листинг] [параметры...] [--]
 ↪ исходный_файл
```

Выполните следующую команду:

```
nasm -o obj.o -f elf -g -l list.lst hello.asm
```

Данная команда скомпилирует исходный файл `hello.asm` в `obj.o` (опция `-o` позволяет задать имя объектного файла, в данном случае `obj.o`), при этом формат выходного файла будет `elf`, и в него будут включены символы для отладки (опция `-g`), кроме того, будет создан файл листинга `list.lst` (опция `-l`).

С помощью команды `ls` проверьте, что файлы были созданы.

Для более подробной информации см. `man nasm`. Для получения списка форматов объектного файла см. `nasm -hf`.

5.4. Компоновщик LD

Как видно из схемы на рис. 5.3, чтобы получить исполняемую программу, объектный файл необходимо передать на обработку компоновщику:

```
ld -m elf_i386 hello.o -o hello
```

С помощью команды `ls` проверьте, что исполняемый файл `hello` был создан.

Компоновщик `ld` не предполагает по умолчанию расширений для файлов, но принято использовать следующие расширения:

- `o` – для объектных файлов;
- `без расширения` – для исполняемых файлов;
- `map` – для файлов схемы программы;
- `lib` – для библиотек.

Ключ `-o` с последующим значением задаёт в данном случае имя создаваемого исполняемого файла.

Выполните следующую команду:

```
ld -m elf_i386 obj.o -o main
```

Какое имя будет иметь исполняемый файл? Какое имя имеет объектный файл из которого собран этот исполняемый файл?

Формат командной строки LD можно увидеть, набрав `ld --help`. Для получения более подробной информации см. `man ld`.

5.4.1. Запуск исполняемого файла

Запустить на выполнение созданный исполняемый файл, находящийся в текущем каталоге, можно, набрав в командной строке:

```
./hello
```

5.5. Задание для самостоятельной работы

1. В каталоге `~/work/arch-pc/lab05` с помощью команды `cp` создайте копию файла `hello.asm` с именем `lab5.asm`
2. С помощью любого текстового редактора внесите изменения в текст программы в файле `lab5.asm` так, чтобы вместо `Hello world!` на экран выводилась строка с вашими фамилией и именем.
3. Оттранслируйте полученный текст программы `lab5.asm` в объектный файл. Выполните компоновку объектного файла и запустите получившийся исполняемый файл.
4. Скопируйте файлы `hello.asm` и `lab5.asm` в Ваш локальный репозиторий в каталог `~/work/study/2022-2023/"Архитектура компьютера"/arch-pc/labs/lab05/`. Загрузите файлы на Github.

5.6. Содержание отчёта

Отчёт должен включать:

- Титульный лист с указанием номера лабораторной работы и ФИО студента.
- Формулировка цели работы.
- Описание результатов выполнения лабораторной работы:
  - описание выполняемого задания;
  - скриншоты (снимки экрана), фиксирующие выполнение заданий лабораторной работы;
  - комментарии и выводы по результатам выполнения заданий.

- Описание результатов выполнения заданий для самостоятельной работы:
  - описание выполняемого задания;
  - скриншоты (снимки экрана), фиксирующие выполнение заданий;
  - комментарии и выводы по результатам выполнения заданий;
  - листинги написанных программ (текст программ).
- Выводы, согласованные с целью работы.

Отчёт по выполнению лабораторной работы оформляется в формате `Markdown`. В качестве отчёта необходимо предоставить отчёты в 3 форматах: `pdf`, `docx` и `md`. А также файлы с исходными текстами написанных при выполнении лабораторной работы программ (файлы `*.asm`). Файлы необходимо загрузить на странице курса в ТУИС в задание к соответствующей лабораторной работе и загрузить на Github.

5.7. Вопросы для самопроверки

1. Какие основные отличия ассемблерных программ от программ на языках высокого уровня?
2. В чём состоит отличие инструкции от директивы на языке ассемблера?
3. Перечислите основные правила оформления программ на языке ассемблера.
4. Каковы этапы получения исполняемого файла?
5. Каково назначение этапа трансляции?
6. Каково назначение этапа компоновки?
7. Какие файлы могут создаваться при трансляции программы, какие из них создаются по умолчанию?
8. Каковы форматы файлов для `nasm` и `ld`?

5.8. Дополнительная информация

5.8.1. Основные возможности текстового редактора mcedit

mcedit — является встроенным текстовым редактором для файлового менеджера Midnight Commander (или mc), который позволяет редактировать различные файлы размером до 64 Мб. К основным возможностям этого редактора можно отнести: копирование блока символов, перемещение, удаление, вырезка, вставка и др.

Например, чтобы создать в текущем каталоге файл hello.asm и начать его редактирование, можно набрать:

```
mcedit hello.asm
```

В общем случае для запуска mcedit может быть использован следующий синтаксис:

```
mcedit [-bcCdfhstVx?] [+число] file
```

Некоторые параметры mcedit пояснены в табл. 5.1.

Таблица 5.1. Некоторые параметры mcedit

Параметр	Действие
+число	переход к указанной числом строке
-b	черно-белая цветовая гамма
-c	цветовой режим ANSI для терминалов без поддержки цвета
-d	отключить поддержку мыши
-V	вывести версию программы

Клавиша F9 служит для вызова меню. В таблице 5.2 приведён список наиболее часто используемых горячих клавиш (Meta — условное обозначение для набора мета-клавиш, обычно это Alt или Esc):

Таблица 5.2. Список наиболее часто используемых горячих клавиш mcedit

Клавиши	Действие
Ctrl + Ins	копировать
Shift + Ins	вставить
Shift + Del	вырезать
Ctrl + Del	удалить выделенный текст
F3	начать выделение текста (повторное нажатие F3 закончит выделение)
Shift + F3	начать выделение блока текста (повторное нажатие F3 закончит выделение)
F5	скопировать выделенный текст
F6	переместить выделенный текст
F8	удалить выделенный текст
Meta + l	переход к строке по её номеру
Meta + q	вставка литерала (непечатного символа)
Meta + t	сортировка строк выделенного текста
Meta + u	выполнить внешнюю команду и вставить в позицию под курсором её вывод
Ctrl + f	занести выделенный фрагмент во внутренний буфер обмена mc (записать во внешний файл)

Клавиши	Действие
Ctrl + k	удалить часть строки до конца строки
Ctrl + n	создать новый файл
Ctrl + s	включить или выключить подсветку синтаксиса
Ctrl + t	выбрать кодировку текста
Ctrl + u	отменить действия
Ctrl + x	перейти в конец следующего слова
Ctrl + y	удалить строку
Ctrl + z	перейти на начало предыдущего слова
Shift + F5	вставка текста из внутреннего буфера обмена `mc` (прочитать внешний файл)
Meta + Enter	диалог перехода к определению функции
Meta + -	возврат после перехода к определению функции
Meta + +	переход вперёд к определению функции
Meta + n	включение/отключение отображения номеров строк
Tab	отодвигает вправо выделенный текст, если выключена опция «Постоянные блоки»
Meta + Tab	отодвигает влево выделенный текст, если выключена опция «Постоянные блоки»
Shift + Стрелки	выделение текста
Meta + Стрелки	выделение вертикального блока
Meta + Shift + -	переключение режима отображения табуляций и пробелов
Meta + Shift + +	переключение режима «Автовыравнивание возвратом каретки»

Клавиши (обычно Alt + Tab или Esc + Tab) служат для автозавершения и завершают слово, на котором находится курсор, используя ранее использовавшиеся в файле.

При помощи команды `man mc` можно получить дополнительную информацию по `mc`.