РОССИЙСКИЙ УНИВЕРСИТЕТ ДРУЖБЫ НАРОДОВ

Факультет физико-математических и естественных наук

Кафедра прикладной информатики и теории вероятностей

Отчет по лабораторной работе №4

дисциплина: Архитектура ЭВМ

Студент: Агзамов Артур Дамирович(1032253528)

Группа: НКАбд-01-25

Содержание

1. **Цель работы стр.4**
2. **Теоретическое введение стр.5**
3. **Выполнение лабораторной работы стр.12**
4. **Выполнение самостоятельной работы стр.17**
5. **Выводы стр.18**

Список иллюстраций

1. *Рис.4.1. Структурная схема ЭВМ стр.5*
2. *Рис.4.2. 64-битный регистр процессора ‘RAX’ 3 стр.6*
3. *Рис.4.3. Процесс создания ассемблерной программы стр.10*
4. *Рис.4.4.lab01 стр.12*
5. *Рис.4.5.lab02 стр.13*
6. *Рис.4.6.lab03 стр.13*
7. *Рис.4.7.lab04 стр.14*
8. *Рис.4.8.lab05 стр.14*
9. *Рис.4.9.lab06 стр.15*
10. *Рис.4.10.lab07 стр.16*
11. *Рис.4.11.lab08 стр.16*
12. *Рис.4.12.sam01 стр.17*
13. *Рис.4.13.sam02 стр.17*
14. *Рис.4.14.sam03 стр.17*
15. *Рис.4.15.sam04 стр.17*

1.Цель работы

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

2. Теоретическое введение

2.1. 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 регистрами или

между регистрами и памятью, преобразование (арифметические или логические операции) данных хранящихся в регистрах.

Рис.4.1. Структурная схема ЭВМ

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.

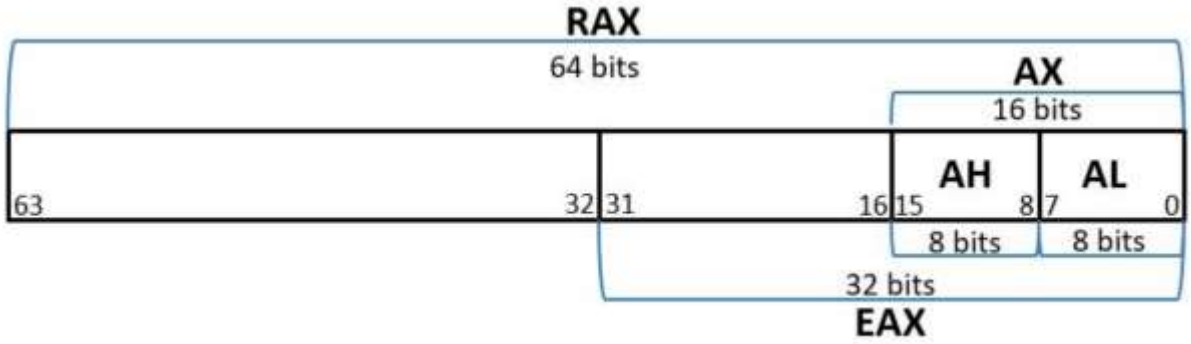

Рис.4.2. 64-битный регистр процессора ‘RAX’

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].

2.3. 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.

2.4. Процесс создания и обработки программы на языке ассемблера

Процесс создания ассемблерной программы можно изобразить в виде следующей схемы (рис. 4.3).

Рис.4.3. 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).

3.Выполнение лабораторной работы

3.1. Программа Hello world!

Рассмотрели пример простой программы на языке ассемблера NASM. Традиционно первая программа выводит приветственное сообщение Hello world! на экран. Создали каталог для работы с программами на языке ассемблера NASM. Перешли в созданный каталог. Создали текстовый файл с именем hello.asm. открыли этот файл с помощью любого текстового редактора, например, gedit. и введите в него соответствующий текст:

```
compagzamov@fedora:~$ mkdir -p ~/work/arch-pc/lab04
compagzamov@fedora:~$ cd ~/work/arch-pc/lab04
compagzamov@fedora:~/work/arch-pc/lab04$ touch hello.asm
compagzamov@fedora:~/work/arch-pc/lab04$ gedit hello.asm
```

Открыть *hello.asm ~/work/arch-pc/lab04

```
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.4.4.lab 01

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.

3.2. Транслятор NASM

NASM превращает текст программы в объектный код. Совершили компиляцию текста программы “Hello World”. Текст программы набран без ошибок, поэтому транслятор преобразует текст программы из файла hello.asm в объектный код, который записывается в файл hello.o. Таким образом, имена всех файлов получаются из имени входного файла и расширения по умолчанию. С помощью команды ls проверили, что объектный файл был создан. Имя объектного файла – hello.o.

Рис.4.5.lab02

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.

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

Выполнили следующую команду:

Рис.4.6.lab03

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

команды ls проверили, что файлы были созданы.

```
compagzamov@fedora:~/work/arch-pc/lab04$ ls
hello  hello.asm  hello.o  list.lst  obj.o
```

Рис.4.7.lab04

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

Рис.4.8.lab05

Рис.4.9.lab06

3.4. Компоновщик LD

Как видно из схемы на рис. 4.3, чтобы получить исполняемую программу, объектный файл необходимо было передать на обработку компоновщику. С помощью команды ls проверили, 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. Формат командной строки LD можно увидеть, набрав ld --help.

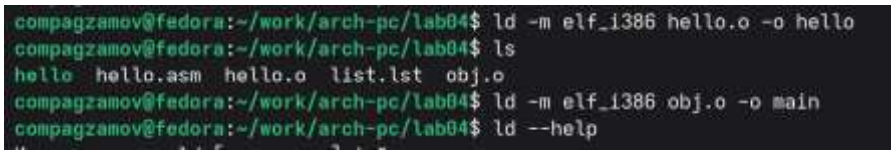

Рис.4.10.lab07

3.5. Запуск исполняемого файла

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

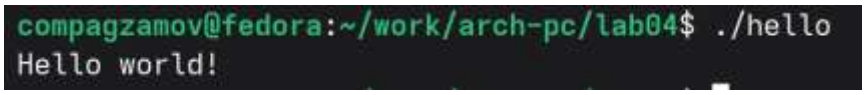

Рис.4.11.lab08

5.Выполнение самостоятельной работы

1. В каталоге ~/work/arch-pc/lab04 с помощью команды cp создали копию файла hello.asm с именем lab4.asm

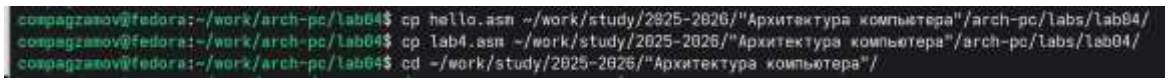

Рис.4.12.sam01

2. С помощью текстового редактора внесли изменения в текст программы в файле lab4.asm так, чтобы вместо Hello world! на экран выводилась строка с моими фамилией и именем.

Рис.4.13.sam02

3. Оттранслировали полученный текст программы lab4.asm в объектный файл. Выполнили компоновку объектного файла и запустили получившийся исполняемый файл.

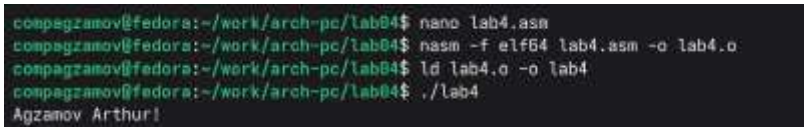

Рис.4.14.sam03

4. Скопировали файлы hello.asm и lab4.asm в Ваш локальный репозиторий в каталог ~/work/ arch-pc/labs/lab04/. Загрузили файлы на Github.

Рис.4.15.sam04

5.Выводы

Были освоены роцедуры компиляции и сборки программ, написанных на ассемблере NASM.

Все задания были выполнены.