

РОССИЙСКИЙ УНИВЕРСИТЕТ ДРУЖБЫ НАРОДОВ
Факультет физико-математических и естественных наук
Кафедра прикладной информатики и теории вероятностей
ОТЧЕТ
ПО ЛАБОРАТОРНОЙ РАБОТЕ №4

дисциплина:

Архитектура компьютера

____ **Студент: ДОНЗО МОРИССАЛА**

Группа: НКАбд-01-24
МОСКВА

2024 г.

1 Цель работы

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

2 Задание

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/lab04
```

Перейдите в созданный каталог:

```
cd ~/work/arch-pc/lab04
```

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
```

Рис. 1: Текст файла hello.asm

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. 2, чтобы получить исполняемую программу, объектный файл необходимо передать на обработку компоновщику:

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



Рис. 2: Процесс создания ассемблерной программы

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:

1. `.o` – для объектных файлов;
2. без расширения – для исполняемых файлов;
3. `.map` – для файлов схемы программы;
4. `.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

3 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. 3).

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

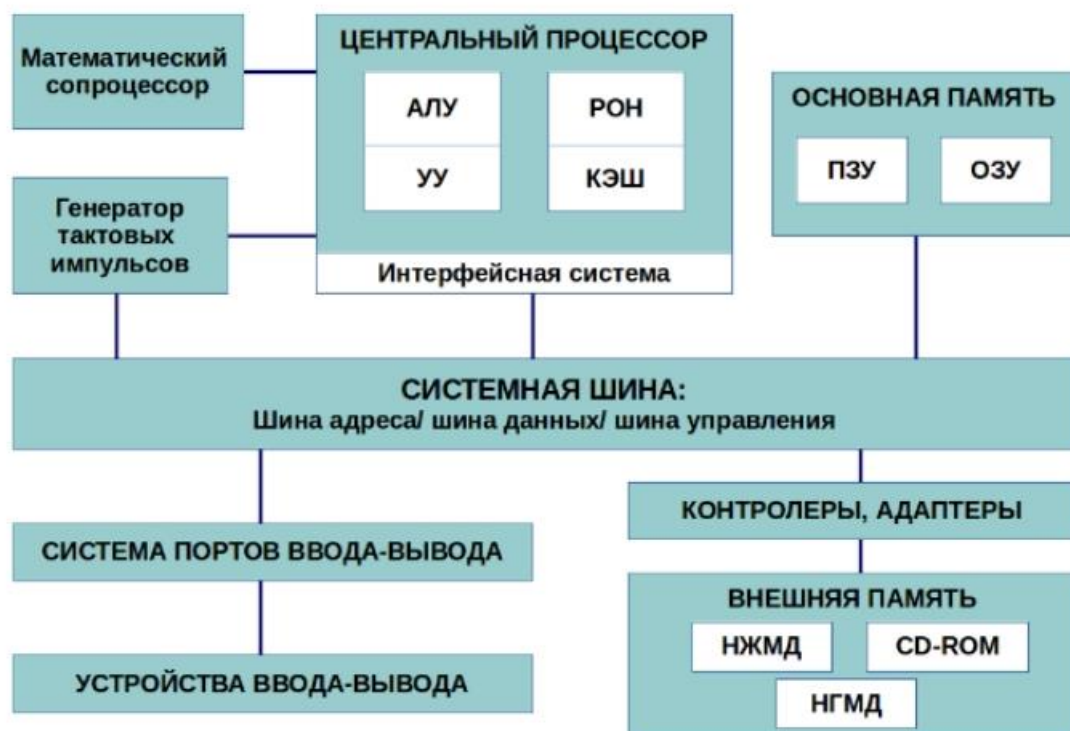


Рис. 3: Структурная схема ЭВМ

Основной задачей процессора является обработка информации, а также организация координации всех узлов компьютера. В состав центрального процессора (ЦП) входят следующие устройства: 1. арифметико-логическое устройство (АЛУ) — выполняет логические и арифметические действия, необходимые для обработки информации, хранящейся в памяти; 2. устройство управления (УУ) — обеспечивает управление и контроль всех устройств компьютера; 3. регистры — сверхбыстрая оперативная память небольшого объема, 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):

1. RAX, RCX, RDX, RBX, RSI, RDI — 64-битные
2. EAX, ECX, EDX, EBX, ESI, EDI — 32-битные
3. AX, CX, DX, BX, SI, DI — 16-битные
4. 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.

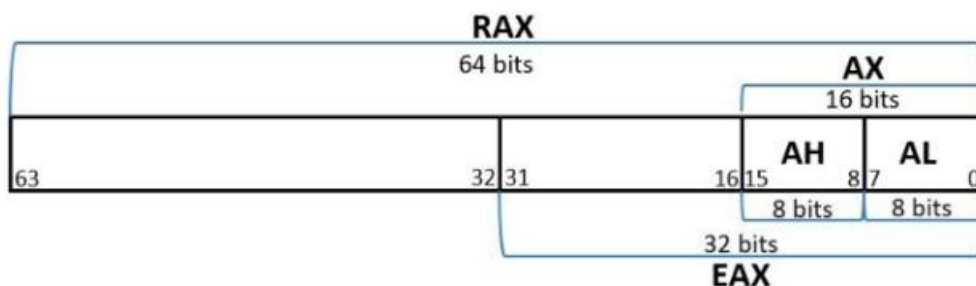


Рис. 4: 64-битный регистр процессора «RAX»

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 раз- делить на: 1. устройства внешней памяти, которые предназначены для долговременного хранения больших объёмов данных (жёсткие диски, твердотельные накопители, магнитные ленты); 2. 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 после довательность стандартных действий, которая называется командным циклом процессора.

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, напи- санные 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 скоро- сти программам, написанным на машинном языке, так как транслятор просто переводит мнемонические обозначения команд в последовательности бит (ну- лей 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: 1. для DOS/Windows: Borland Turbo Assembler (TASM), Microsoft Macro Assembler (MASM) и Watcom assembler (WASM); 2. для GNU/Linux: gas (GNU Assembler), использующий

AT&T-синтаксис, в отличие от большинства других популярных ассемблеров, которые используют Intel-синтаксис.

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. 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. 2).



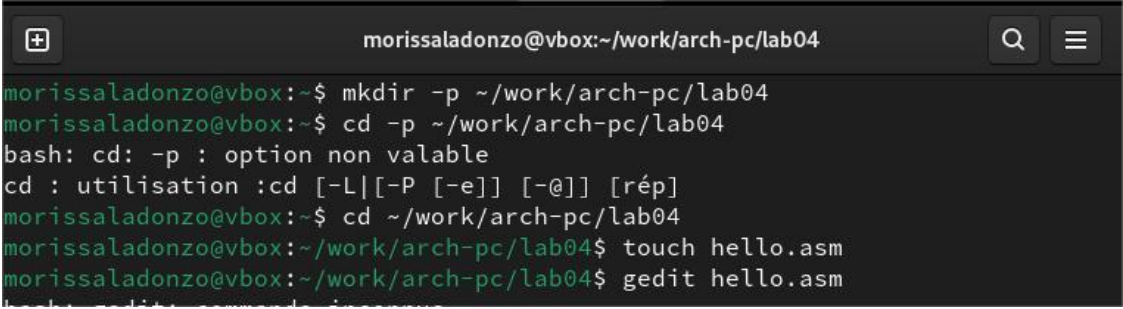
Рис. 2: Процесс создания ассемблерной программы

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).

4 Выполнение лабораторной работы

Первым делом я перешёл в каталог с 5-й лабораторной работой, создал файл `hello.asm` и ввёл туда текст с рисунка №1 (рис.5)



```
morissaladonzo@vbox:~/work/arch-pc/lab04
morissaladonzo@vbox:~$ mkdir -p ~/work/arch-pc/lab04
morissaladonzo@vbox:~$ cd -p ~/work/arch-pc/lab04
bash: cd: -p : option non valable
cd : utilisation :cd [-L|[-P [-e]] [-@]] [rép]
morissaladonzo@vbox:~$ cd ~/work/arch-pc/lab04
morissaladonzo@vbox:~/work/arch-pc/lab04$ touch hello.asm
morissaladonzo@vbox:~/work/arch-pc/lab04$ gedit hello.asm
```

```
1 ; hello.asm
2
3 SECTION .data ; Начало секции данных
4
5     hello:      DB 'Hello world!',10 ; 'Hello world!' плюс
6                                     ; символ перевода строки
7     helloLen: EQU $-hello ; Длина строки hello
8
9 SECTION .text ; Начало секции кода
10    GLOBAL _start
11
12
13 _start: ; Точка входа в программу
14     mov eax,4 ; Системный вызов для записи (sys_write)
15
16     mov ebx,1 ; Описатель файла '1' - стандартный вывод
17
18     mov ecx,hello ; Адрес строки hello в ecx
19
20     mov edx,helloLen ; Размер строки hello
21
22     int 80h ; Вызов ядра
23
24
25     mov eax,1 ; Системный вызов для выхода (sys_exit)
26     mov ebx,0 ; Выход с кодом возврата '0' (без ошибок)
27     int 80h ; Вызов ядра
```

Рис. 5: Создание файла hello.asm

После этого я преобразовал текст программы с помощью транслятора в объектный код, который был записан в файл hello.o (рис.6).

```
morissaladonzo@vbox:~/work/arch-pc/lab04$ gedit hello.asm
morissaladonzo@vbox:~/work/arch-pc/lab04$ nasm -f elf hello.asm
morissaladonzo@vbox:~/work/arch-pc/lab04$ ls
hello.asm  hello.o
morissaladonzo@vbox:~/work/arch-pc/lab04$
```

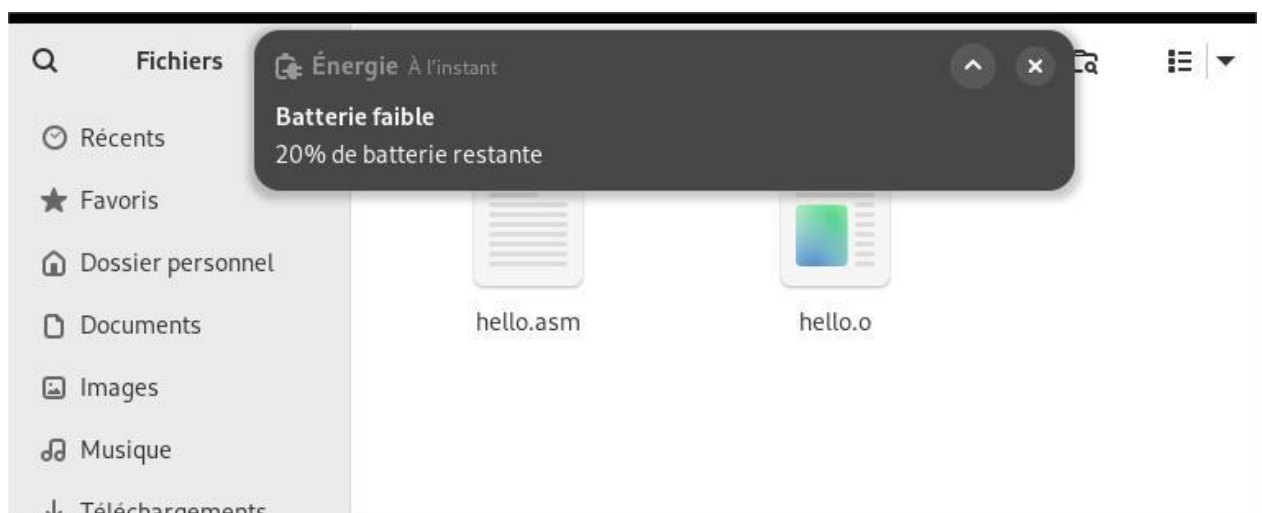


Рис. 6: Создание объектного кода

Затем я скомпилировал исходный файл `hello.asm` в `obj.o` и создал файл листинга `list.lst` (рис.7).

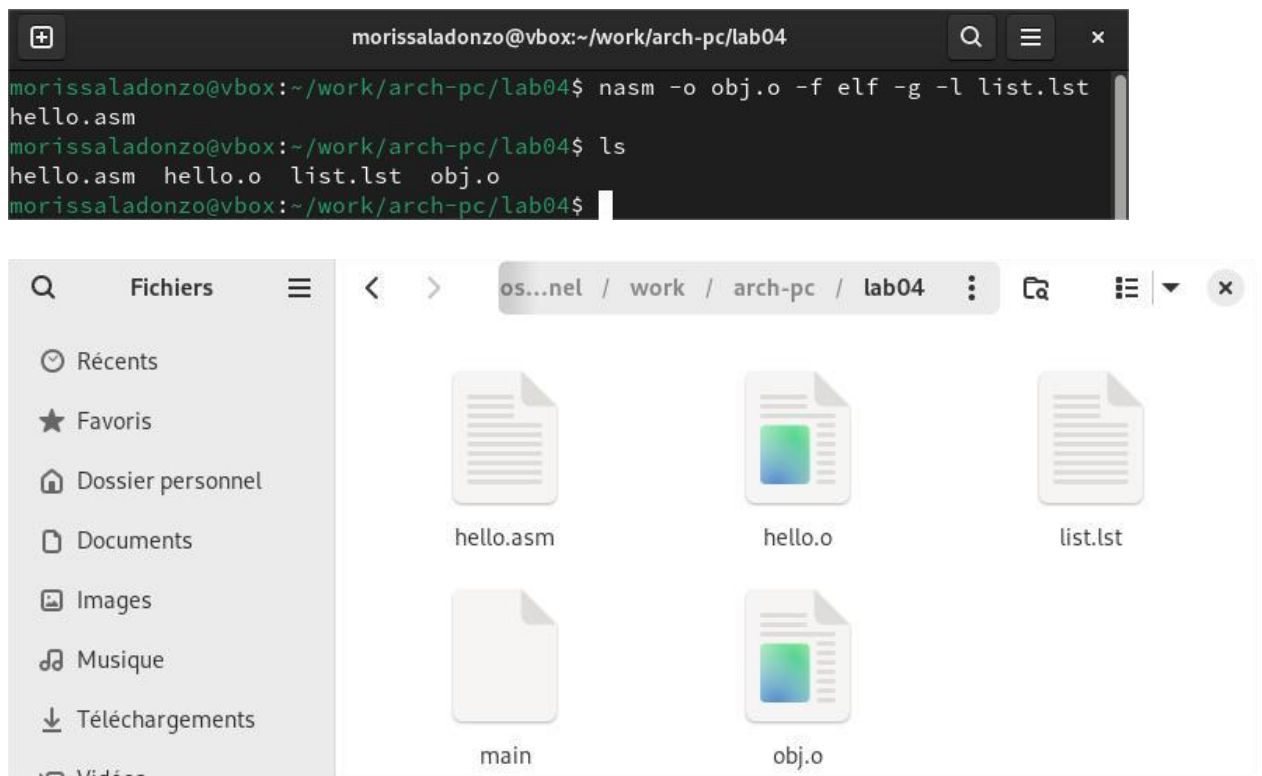
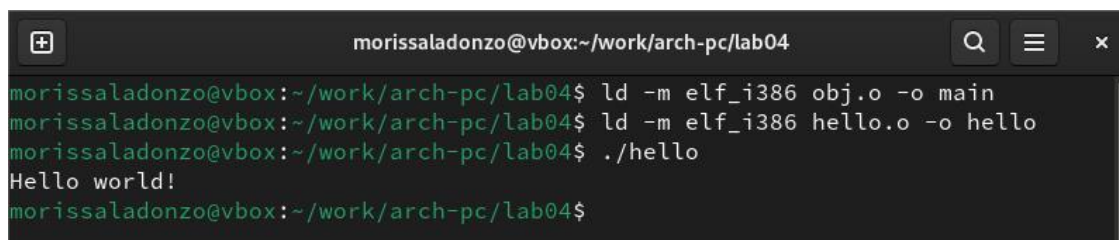


Рис. 7: Компиляция исходного файла

Далее я передал исполняемую программу на обработку компоновщику, задал имя создаваемого исполняемого файла и запустил его (рис.8).



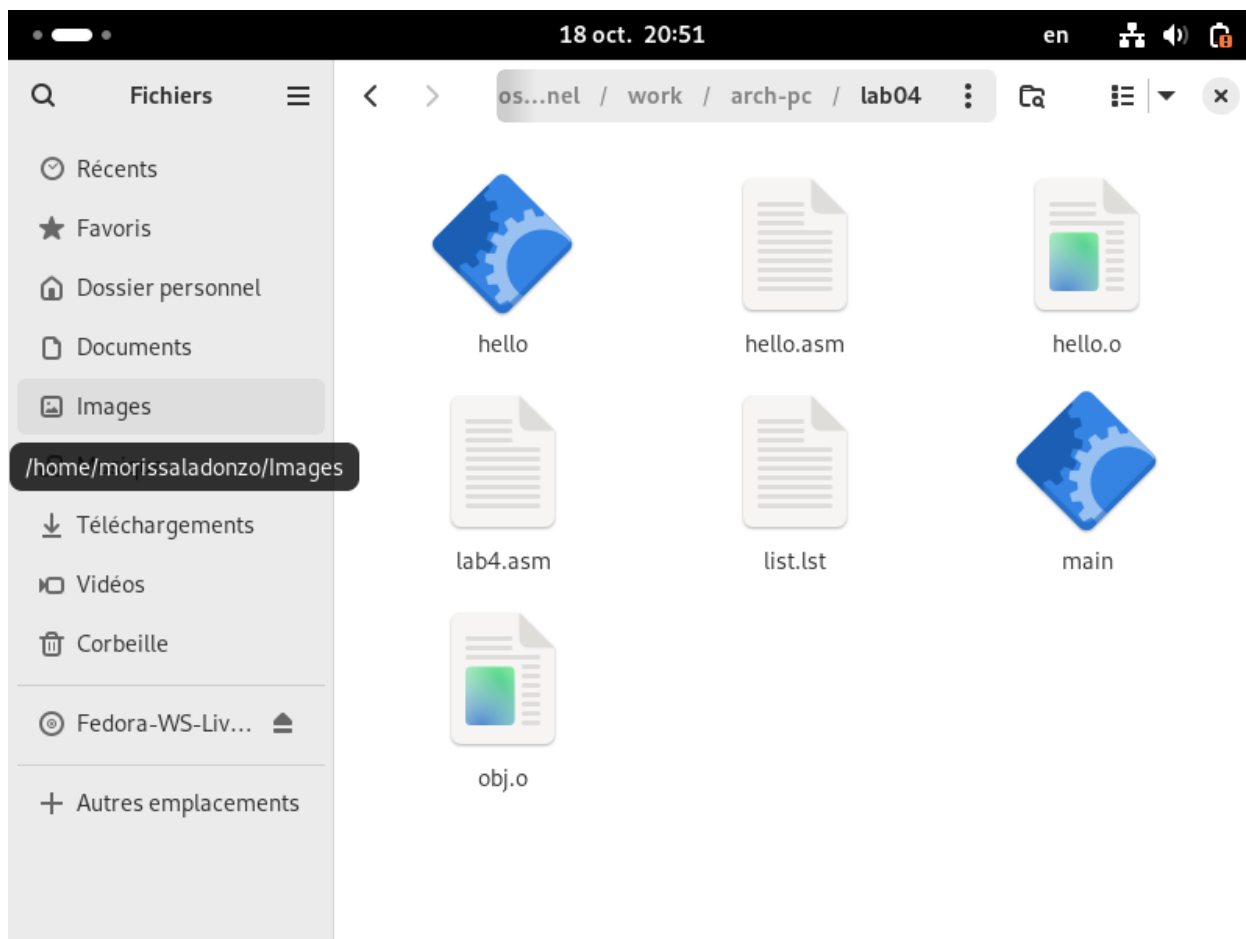


Рис. 8: Работа программы

5 Выводы

В результате выполнения данной лабораторной работы я освоил процедуру компиляции и сборки программ, написанных на языке NASM.

6 Задание для самостоятельной работы

```
morissaladonzo@vbox:~/work/arch-pc/lab04$ cp hello.asm lab4.asm
morissaladonzo@vbox:~/work/arch-pc/lab04$ gedit lab4.asm
```

```
18 oct. 22:36
morissaladonzo@vbox:~/work/arch-pc/lab04$ ./lab4
Донзо Мориссала
morissaladonzo@vbox:~/work/arch-pc/lab04$
```

18 oct. 22:39en

Ouvrirlab4.asmEnregistrer

~/work/arch-pc/lab04

```
1 ; lab4.asm
2
3 SECTION .data ; Начало секции данных
4
5     lab4:      DB ' Донзо Мориссала',10 ; 'Донзо Мориссала' плюс
6                ; символ перевода строки
7     lab4Len: EQU $-lab4 ; Длина строки hello
8
9 SECTION .text ; Начало секции кода
10    GLOBAL _start
11
12
13 _start: ; Точка входа в программу
14     mov eax,4 ; Системный вызов для записи (sys_write)
15
16     mov ebx,1 ; Описатель файла '1' - стандартный вывод
17
18     mov ecx,lab4 ; Адрес строки hello в ecx
19
20     mov edx,lab4Len ; Размер строки hello
21
22     int 80h ; Вызов ядра
23
24
25     mov eax,1 ; Системный вызов для выхода (sys_exit)
26     mov ebx,0 ; Выход с кодом возврата '0' (без ошибок)
27     int 80h ; Вызов ядра
```

Texte brut ▾ Largeur des tabulations : 8 ▾ Lig 24, Col 9INS

Список литературы

Лабораторная работа №4 (Архитектура ЭВМ).