

Министерство науки и высшего образования Российской Федерации  
Федеральное государственное бюджетное образовательное учреждение  
высшего образования

ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ  
УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)  
Кафедра безопасности информационных систем (БИС)

## ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ АССЕМБЛЕР

Отчет по лабораторной работе №2  
по дисциплине «Системное программирование»

Студент гр.748

\_\_\_\_\_А.А. Мухачева

\_\_\_\_\_.\_\_\_\_.2022

Принял

Преподаватель каф. КИБЭВС

\_\_\_\_\_Е.О. Калинин

\_\_\_\_\_.\_\_\_\_.2022

Томск 2022

## **Введение**

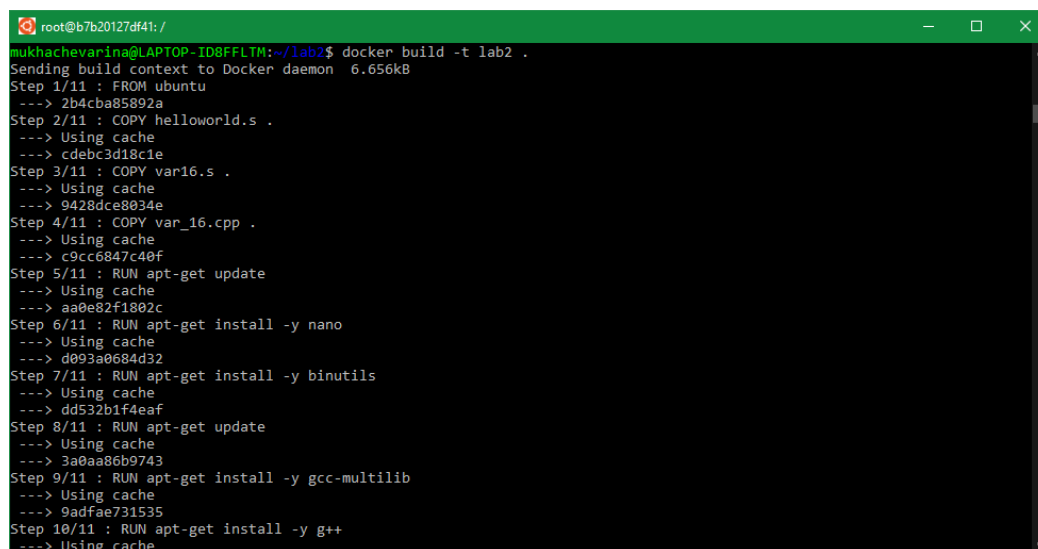
Целью данной лабораторной работы является знакомство со структурой программы на языке Ассемблер, разновидностями и назначением сегментов, способами организации простых и сложных типов данных, изучение формата и правил работы с транслятором GAS и средствами создания программ на Ассемблере для ОС Linux.

## 1 Ход работы

Запуск программ, работа с отладчиком, сравнение по времени выполнения и по памяти производились в Docker. Поэтому был написан dockerfile, в котором в качестве образа внутри контейнера использовалась Ubuntu, скопированы три программы (hello world на ассемблере, две программы по варианту – на с++ и на ассемблере). Также установлены пакеты для работы с ассемблером и языком высокого уровня, отладчик gdb и текстовый редактор nano.

```
FROM ubuntu
COPY helloworld.s .
COPY var16.s .
COPY var_16.cpp .
RUN apt-get update
RUN apt-get install -y nano
RUN apt-get install -y binutils
RUN apt-get update
RUN apt-get install -y gcc-multilib
RUN apt-get install -y g++
RUN apt-get install -y gdb
```

Далее была выполнена команда `docker build -t lab2 .` для создания образа на основе Dockerfile. Результат представлен на рисунке 1.1.



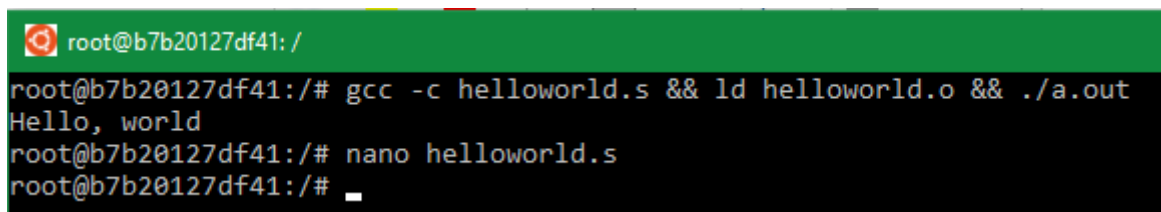
```
root@b7b20127df41: /
mukhachevarina@LAPTOP-ID8FFLTM:~/lab2$ docker build -t lab2 .
Sending build context to Docker daemon  6.656kB
Step 1/11 : FROM ubuntu
--> 2b4cba85892a
Step 2/11 : COPY helloworld.s .
--> Using cache
--> cdebc3d18c1e
Step 3/11 : COPY var16.s .
--> Using cache
--> 9428dce8034e
Step 4/11 : COPY var_16.cpp .
--> Using cache
--> c9cc6847c40f
Step 5/11 : RUN apt-get update
--> Using cache
--> aa0e82f1802c
Step 6/11 : RUN apt-get install -y nano
--> Using cache
--> d093a0684d32
Step 7/11 : RUN apt-get install -y binutils
--> Using cache
--> dd532b1f4eaf
Step 8/11 : RUN apt-get update
--> Using cache
--> 3a0aa86b9743
Step 9/11 : RUN apt-get install -y gcc-multilib
--> Using cache
--> 9adf4e731535
Step 10/11 : RUN apt-get install -y g++
--> Using cache
```

Рисунок 1.1 – Создание образа

Далее посредством команды `docker images` был проверен список всех образов, для того чтобы узнать ID созданного образа. И на его основе был запущен контейнер (команда `docker run -it lab2`).

Внутри контейнера первым делом была написана программа на языке ассемблер, результатом выполнения которой являлась строка «Hello, world» (рисунок 1.2).

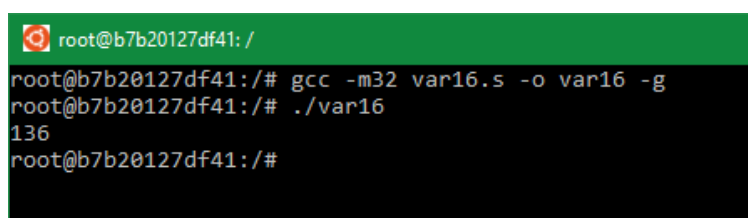
Команда для запуска: `gcc -c helloworld.s && ld helloworld.o && ./a.out`



```
root@b7b20127df41: /  
root@b7b20127df41:/# gcc -c helloworld.s && ld helloworld.o && ./a.out  
Hello, world  
root@b7b20127df41:/# nano helloworld.s  
root@b7b20127df41:/#
```

Рисунок 1.2 – Выполнение hello-world на ассемблере

С помощью последовательного выполнения команд `gcc -m32 var16.s -o var16 -g` и `./var16` был скомпилирован код, и запущен файл программы по варианту на ассемблере. Результат представлен на рисунке 1.3. Программа, написанная согласно индивидуальному варианту на ассемблере, представлена в приложении А.



```
root@b7b20127df41: /  
root@b7b20127df41:/# gcc -m32 var16.s -o var16 -g  
root@b7b20127df41:/# ./var16  
136  
root@b7b20127df41:/#
```

Рисунок 1.3 – Выполнение программы по варианту на ассемблере

Далее на высоком языке программирования была написана программа, выполняющая те же действия, а именно находит сумму элементов массива, которые были сдвинуты логически вправо, в том случае если они имели 1 в 7 бите (рисунок 1.4). Код программы представлен в приложении Б.

Команды: `g++ var_16.cpp -o var_16; ./var_16`

```
root@b7b20127df41: /
root@b7b20127df41:/# g++ var_16.cpp -o var_16
root@b7b20127df41:/# ./var_16
136
root@b7b20127df41:/# _
```

Рисунок 1.4 – Выполнение программы на C++

По итогу обе программы получают одинаковый результат. Далее был запущен отладчик (рисунок 1.5).

```
root@b7b20127df41: /
root@b7b20127df41:/ls
a.out  boot  etc      helloworld.s  lib  lib64  media  opt  root  sbin  sys  usr  var16  var_16
bin    dev    helloworld.o  home         lib32 libx32 mnt    proc run  srv   tmp  var  var16.s  var_16.cpp
root@b7b20127df41:/# gdb var16
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software; you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from var16...
(gdb) _
```

Рисунок 1.5 – Запуск GDB

Далее, при помощи команды `break main`, была поставлена точка останова в функции `main` (`break main`) и запущена на исполнение с помощью команды `run` (рисунок 1.6).

```
Выбрать root@b7b20127df41: /
(gdb) break main
Breakpoint 1 at 0x5664b1ad: file var16.s, line 11.
(gdb) run
Starting program: /var16
warning: Error disabling address space randomization: Operation not permitted

Breakpoint 1, main () at var16.s:11
11      xorl %eax, %eax
```

Рисунок 1.6 – Точка останова

Далее была выполнена команда `info reg`, которая отображает содержимое регистров (рисунок 1.7).

```
(gdb) info reg
eax             0xf7f79088      -134770552
ecx             0x54744136      1416905014
edx             0xffeabfb4      -1392716
ebx             0x0              0
esp             0xffeabf8c      0xffeabf8c
ebp             0x0              0x0
esi             0xf7f77000      -134778880
edi             0xf7f77000      -134778880
eip             0x565d01ad      0x565d01ad <main>
eflags          0x246           [ PF ZF IF ]
cs              0x23            35
ss              0x2b            43
ds              0x2b            43
es              0x2b            43
fs              0x0              0
gs              0x63            99
(gdb)
```

Рисунок 1.7 – Содержимое регистров

Далее было осуществлено передвижение по командам в программе с помощью `s`. С помощью команды `i r eax` было определено состояние регистра. Результат представлен на рисунках 1.8 и 1.9.

```
root@b7b20127df41: /
15          shrl $1, (%ebx) /*сдвиг на 1 разряд*/
(gdb) s
16          addl (%ebx), %eax /*сложили*/
(gdb) s
17          addl $4, %ebx
(gdb) s
18          jmp check
(gdb) s
27          cmpl $end, %ebx /*последний ди элемент*/
(gdb) s
28          jne sdvig_if_7
(gdb) s
30          pushl %eax
(gdb) s
31          pushl $vyvod
(gdb) s
32          call printf
(gdb) s
0xf7dded30 in printf () from /lib32/libc.so.6
(gdb) s
Single stepping until exit from function printf,
which has no line number information.
136
check () at var16.s:33
33          addl $8, %esp
```

Рисунок 1.8 – Передвижение по программе

```
(gdb) i r eax
eax          0x4          4
(gdb) _
```

Рисунок 1.9 – Предоставление информации о состоянии регистра

В конце была выполнена команда `disassemble`, было произведено дизассемблирование кода (рисунок 1.10).

```
root@b7b20127df41: /
(gdb) disassemble
Dump of assembler code for function check:
0x565d01d2 <+0>:    cmp     $0x565d301c,%ebx
0x565d01d8 <+6>:    jne     0x565d01bf <sdvig_if_7>
0x565d01da <+8>:    push   %eax
0x565d01db <+9>:    push   $0x565d3008
0x565d01e0 <+14>:   call   0xf7dded30 <printf>
0x565d01e5 <+19>:   add     $0x8,%esp
=> 0x565d01e8 <+22>:   mov     $0x0,%eax
0x565d01ed <+27>:   ret
0x565d01ee <+28>:   xchg    %ax,%ax
End of assembler dump.
(gdb) _
```

Рисунок 1.10 – Дизассемблирование кода

Далее для программы, написанной на С, была выставлена аналогичная точка останова в функции `main` и проведено дизассемблирование выбранного участка кода (рисунки 1.11-1.13).

```
root@b7b20127df41: /
root@b7b20127df41: /# gdb var_16
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from var_16...
(No debugging symbols found in var_16)
(gdb) run
Starting program: /var_16
warning: Error disabling address space randomization: Operation not permitted
136
[Inferior 1 (process 38) exited normally]
(gdb) _
```

Рисунок 1.11 – Точка останова

```
root@b7b20127df41: /
(gdb) break main
Breakpoint 1 at 0x55bc964f0149
(gdb) run
Starting program: /var_16
warning: Error disabling address space randomization: Operation not permitted

Breakpoint 1, 0x000056378fd96149 in main ()
(gdb) info reg
rax            0x56378fd96149      94796636578121
rbx            0x56378fd96200      94796636578304
rcx            0x56378fd96200      94796636578304
rdx            0x7ffeddbf40e8      140732618719464
rsi            0x7ffeddbf40d8      140732618719448
rdi            0x1              1
rbp            0x0              0x0
rsp            0x7ffeddbf3fe8      0x7ffeddbf3fe8
r8             0x0              0
r9             0x7fda6a938d50      140576067652944
r10            0xffffffff      4294967295
r11            0x0              0
r12            0x56378fd96060      94796636577888
r13            0x7ffeddbf40d0      140732618719440
r14            0x0              0
r15            0x0              0
rip            0x56378fd96149      0x56378fd96149 <main>
eflags         0x246             [ PF ZF IF ]
cs             0x33             51
ss             0x2b             43
ds             0x0              0
es             0x0              0
```

Рисунок 1.12 – Содержимое регистров

```
root@b7b20127df41: /
(gdb) disassemble
Dump of assembler code for function main:
=> 0x000055d13ab26149 <+0>:      endbr64
0x000055d13ab2614d <+4>:      push    %rbp
0x000055d13ab2614e <+5>:      mov     %rsp,%rbp
0x000055d13ab26151 <+8>:      sub     $0x10,%rsp
0x000055d13ab26155 <+12>:     movl    $0x0,-0x4(%rbp)
0x000055d13ab2615c <+19>:     movl    $0x0,-0x8(%rbp)
0x000055d13ab26163 <+26>:     cmpl    $0x3,-0x8(%rbp)
0x000055d13ab26167 <+30>:     jg      0x55d13ab261db <main+146>
0x000055d13ab26169 <+32>:     mov     -0x8(%rbp),%eax
0x000055d13ab2616c <+35>:     cltq
0x000055d13ab2616e <+37>:     lea     0x0(,%rax,4),%rdx
0x000055d13ab26176 <+45>:     lea     0x2e93(%rip),%rax      # 0x55d13ab29010 <massiv>
0x000055d13ab2617d <+52>:     mov     (%rdx,%rax,1),%eax
0x000055d13ab26180 <+55>:     and     $0x80,%eax
0x000055d13ab26185 <+60>:     test    %eax,%eax
0x000055d13ab26187 <+62>:     je      0x55d13ab261d5 <main+140>
0x000055d13ab26189 <+64>:     mov     -0x8(%rbp),%eax
0x000055d13ab2618c <+67>:     cltq
0x000055d13ab2618e <+69>:     lea     0x0(,%rax,4),%rdx
0x000055d13ab26196 <+77>:     lea     0x2e73(%rip),%rax      # 0x55d13ab29010 <massiv>
0x000055d13ab2619d <+84>:     mov     (%rdx,%rax,1),%eax
0x000055d13ab261a0 <+87>:     sar     %eax
0x000055d13ab261a2 <+89>:     mov     %eax,%ecx
0x000055d13ab261a4 <+91>:     mov     -0x8(%rbp),%eax
0x000055d13ab261a7 <+94>:     cltq
0x000055d13ab261a9 <+96>:     lea     0x0(,%rax,4),%rdx
0x000055d13ab261b1 <+104>:    lea     0x2e58(%rip),%rax      # 0x55d13ab29010 <massiv>
0x000055d13ab261b8 <+111>:    mov     %ecx,(%rdx,%rax,1)
```

Рисунок 1.13 – Дизассемблирование кода

Далее было выполнено сравнение по времени выполнения программ (рисунок 1.14) и по объему (рисунок 1.15).

Команды: `time ./var16; time ./var_16`



```
root@b7b20127df41: /
root@b7b20127df41:/# time ./var16
136

real    0m0.005s
user    0m0.003s
sys     0m0.000s
root@b7b20127df41:/# time ./var_16
136

real    0m0.007s
user    0m0.005s
sys     0m0.001s
root@b7b20127df41:/#
```

Рисунок 1.14 – Сравнение по времени

Команды: `ls -s -h var16`; `ls -s -h var_16`

```
Выбрать root@b7b20127df41: /
root@b7b20127df41:/# ls -s -h var16
16K var16
root@b7b20127df41:/# ls -s -h var_16
20K var_16
root@b7b20127df41:/# _
```

Рисунок 1.15 – Сравнение по объему

Было выяснено, что дизассемблированная программа, написанная на C++ больше по объему, чем код, написанный на Assembler.

## **Заключение**

В результате выполнения данной лабораторной работы была изучена структура программ на языке Assembler, изучены форматы и правила работы с компилятором GCC и отладчиком GDB. Так же были написаны программы по индивидуальному заданию на языках Assembler и C++. В результате работы были получены выводы о том, что программа, реализованная на Assembler, выполняется быстрее и имеет меньший объём, нежели программа, написанная на C++.

## Приложение А

(обязательное)

### Код программы на Assembler

```
.data
vyvod:
    .string "%d\n"
massiv:
    .long 128, 10, 12, 144 /*определение содержимого массива*/
end:

.text
.global main

main:
xorl $eax, %eax /*eax - будет записана сумма*/
movl $massiv, %ebx /*ebx - адрес текущего элемента массива*/
jmp check

sum_massiv:
shrl $1, (%ebx) /* сдвиг логически вправо на 1 разряд*/
addl (%ebx), %eax /* складываем результат в %eax*/
addl $4, %ebx
jne check

sdvig_if_7:
movl (%ebx), %ecx
andl $128, %ecx
cmpl $128, %ecx /* проверка наличия 1 в 7 бите*/
je sum_massiv /* переход если равно*/
addl $4, %ebx /*выделяем место и переходим на следующий элемент, путём добавления бит*/

check:
cmpl $end, %ebx /*последний ли элемент (адрес)*/
je sdvig_if_7 /*если это последний элемент, проверяем на наличие 1*/

pushl %eax
pushl $vyvod
call printf
addl $8, %esp
movl $0, %eax
ret
```

## Приложение Б

(обязательное)

Код программы на C++

```
#include <stdio.h>
#include <stdlib.h>
int array [5] = {67,99,3,58, 114};
int main ()
{
    int i;
    int summ=0;
    for (i=0; i!=5; i++)
    { if ((array[i]&128)==128)
      {array[i]=array[i]>
      >1;
      summ=summ+array[i];
      }}
    printf ("%i\n",
    sum);return 0;
}
```