

Введение в архитектуру ЭВМ

Лекция 2. Регистры. Символьные имена. Задача первой лабораторной работы.

План лекции

Регистры процессора, видеообзор

Символьные имена

Синтаксис AT&T

Синтаксисы AT&T и Intel – основные отличия

Метки - символьные имена с адресными значениями. Процесс ассемблирования

Символьные имена с произвольными значениями

Программа `Sym-names.S` – пример процесса ассемблирования

Таблица символов исполняемого файла `Sym-names`

Вывод таблицы символов по команде `nm`

Задача первой лабораторной работы

Исходный файл `task1.c`

Исходный файл `task1.S`

Иллюстрация работы команд `idivl` (описание ее на стр. 101).

Выводы

Регистры процессора, видеообзор

Символьные имена

Синтаксис AT&T

AT&T используется по умолчанию.

Т.к. почти все документы по 80386 используют синтаксис Intel то после директивы `.intel_syntax as` начинает использовать этот синтаксис, а после директивы `.att_syntax` переходит обратно.

Синтаксически символьное имя это один или несколько символов из множества прописных и строчных букв английского, цифр и трех знаков: «_», «.» и «\$». Символьное имя не может начинаться с цифры. Прописные и строчные буквы считаются разными. Ограничений на длину не существует.

Символьные имена ограничиваются знаками не из этого множества (например пробел, табуляция) или началом файла (так как исходная программа должна кончатся новой строкой, конец файла не может считаться ограничителем символа).

NBVB. Символьное имя имеет двоичное значение (как правило 32-битовое) и тип – `t`, `d`, `b` или `a` (возможны также `T`, `D`, `B`, `A`), которые определяются в процессе ассемблирования по названию секции где это имя определено.

Синтаксисы AT&T и Intel – основные отличия.

- Перед непосредственными операндами в AT&T стоит символ "\$".
- Перед операндами регистра в AT&T стоит символ "%".
- В синтаксисе AT&T и Intel используется противоположный порядок следования операндов источника и приемника.
- В синтаксисе AT&T размер операндов памяти определяется по

последнему символу кода операции. Суффиксы 'b', 'w', 'l' и 'q' указывают на адреса байта (8 бит), слова (16 бит), двойного слова (32 бита) и четверного слова (64 бита).

Другие отличия синтаксисов представлены в руководстве «The gnu Assembler Using as (GNU Binutils).

Метки - символьные имена с адресными значениями. Процесс ассемблирования

Определение. Метка — это символьное имя за которым без пробела следует символ «:».

Значениями меток являются адреса в ОП элементов объектного, а после работы ld — исполняемого файла — прежде всего адреса данных и адреса передач управления в операндах команд.

Метки получают значение в процессе ассемблирования при последовательном чтении строк исходного файла с переводом их в двоичные образы и размещением последних в объектном файле.

При этом в каждой из секций, .data, .text .bss и других ассемблер ведет независимые 32-разрядные целые счетчики размещения с нулевыми начальными значениями.

NB. Ассемблер присваивает метке значение счетчика размещения в объектном файле, которое он имел ДО обработки строки исходного кода, в которой метка определена. После этого формируется и размещается в объектном файле двоичный образ этой строки и счетчик увеличивается на занимаемое этим образом количество байтов.

NB. При использовании меток как операндов команд ассемблера

при формировании соответствующей команды ЯКЦП в процессорном формате значение метки помещается в поле «Смещение».

Значения меток называются перемещаемыми (relocatable), т. е. перемещаются (настраиваются) редактором связей ld для выполнения в конкретном блоке ОП, выделяемым загрузчиком.

Текущее значение счетчика размещения по которому выполняется ассемблирование всегда доступно программисту с помощью специального символьного имени «.». Так директива

```
Sch_razm_1: .long .
```

присвоит двойному слову, на которое указывает метка Sch_razm_1:, ее же адресное значение, как это, например, показано в фрагменте файла листинга:

```
11 0005 05000000 Sch_razm_1: .long .
```

NBNB. Т.о. значением метки в секции, в том числе в секциях .text, .data, .bss объектного файла, является количество байтов ОП от начала секции до адреса, соответствующего метке.

Также для этого значения можно использовать более общий термин, применяемый Intel – перемещение (offset).

Определение. Перемещение байта в ОП – это разность между его адресом в некотором блоке ОП и адресом первого байта этого блока.

Адреса байтов	Текст секций
	.data
0000	
0001	
...	
003F	Sch_razm_0: .long .
...	

	.text
0000	
0001	
...	
0A2E	NextDigit:
...	

Согласно этой иллюстрации в блоке ОП секции `.data` значением и перемещением символьного имени – метки `Sch_razm_0`: является `003F`, а в блоке ОП секции `.text` значением и перемещением символьного имени – метки `NextDigit`: - `0A2E`.

NBNB. Итак, наличие символа «:» в конце символьного имени заставляет ассемблер определять его значение в процессе ассемблирования с помощью счетчика размещения.

Символьные имена с произвольными значениями.

Программист может присвоить символьному имени в конце которого отсутствует символ «:» любое значение с помощью эквивалентных конструкций:

```
<СИМВОЛЬНОЕ ИМЯ> = <ВЫРАЖЕНИЕ>
.set <СИМВОЛЬНОЕ ИМЯ>, <ВЫРАЖЕНИЕ>
```

где `<выражение>` определяет числовую константу или перемещение в некоторой секции.

Примечание.

Тип такого символьного имени зависит от типов членов выражения. Если же выражение определяет числовую константу то для этого символьного имени в таблице символов указывается тип `*ABS*` - «абсолютный» тип, (в выводе утилиты `nm - a`) – от примененного в

описании as термина «absolute number» . Чтобы избежать неясностей, связанных с русским термином «абсолютное значение» , мы будем называть этот тип «ПОСТОЯННЫЙ» .

Авторы описания имели в виду, что при построении исполняемого файла редактор связей изменяет начальные адреса всех секций и, следовательно, адресные значения определенных в них символьных имен, прежде всего меток, определенных в этих секциях . Такие имена называются перемещаемыми. Но символьные имена типа «ПОСТОЯННЫЙ» редактор связей не изменяет. Ясно, что если определить символьное имя как, например:

```
Buf_length = 50
```

то его значение не требует перемещения и не будет изменяться редактором связей

Выражение строится из определенных в исходном тексте символьных имен, констант и знаков арифметических и логических операций. Мы будем использовать только простейшие выражения.

Такие символьные имена можно применять, например, для исключения т. н. «магических чисел», вычисления длин структур данных, например строк, для других целей. Ниже будут рассмотрены примеры.

Программа Sym-names.S – пример процесса ассемблирования

GAS LISTING Sym-names.S

page 1

```

2
3      .bss
4      buf_Length0 = 50 # постоянное
выражение
5      .lcomm c, 1      # директива
выделения
6      .lcomm buf, buf_Length0 # ОП в bss
7
8      .data
9
10     buf_Length1 = buf_Length0 + 30 #
постоянное выражение
11
12     Sch_razm_0:
13 0000 00000000      .long .
14     Bait:
15 0004 F0           .byte 0xF0
16
17 0005 000000      .align 4 # на алрес ОП кратный 4
18     Buf0:
19 0008 3200        .word buf_Length0
20     Buf1:
21 000a 5000        .word buf_Length1
22     Summ_buf:
23 000c 8200        .word  buf_Length0 + buf_Length1 #
постоянное выражение1
24
25 000e 0000        .align 4
26     Sch_razm_1:
27 0010 10000000    .long .

```

```

28 Slowo:
29 0014 CDAB .word 0xABCD
30
31 0016 0000 .align 4
32 Sch_razm_2:
33 0018 18000000 .long .
34 Dvoinoe_Slovo:
35 001c DCFE4512 .long 0x1245FEDC
36
37 Sch_razm_3:
38 0020 20000000 .long .
39
40 Adres_Stroki = .
41 # перемещаемое значение выражения
42
43 Adr_2_baita_Stroki = . + 1
44 # перемещаемое значение выражения
45
46 Stroka:
47 0024 31323334 .ascii "1234 ABCD АБВГ "
47 20414243
47 4420D090
47 D091D092
47 D09320
48
49 # вычисляем длину строки
50
51 Str_Len = . - Stroka # постоянное

```

выражение

52

```

53                               Adr_posledn_baita_Stroki = . - 1
GAS LISTING Sym-names.S                               page 2

54                               # перемещаемое значение выражения
55
56                               Adres_Stroki_v_0P:
57 0037 24000000                .long Adres_Stroki
58
59
60
61
62                               Adr_posledn_baita_Stroki_v_0P:
63 003b 36000000                .long Adr_posledn_baita_Stroki
64
65                               Dlina_Stroki_v_0P:
66 003f 13000000                .long Str_Len
67
68                               .end

```

```

GAS LISTING Sym-names.S                               page 3

```

DEFINED SYMBOLS

```

Sym-names.S:4                *ABS*:00000032 buf_Length0
                               .bss:00000000 c
Sym-names.S:6                .bss:00000008 buf
Sym-names.S:10               *ABS*:00000050 buf_Length1
Sym-names.S:12               .data:00000000 Sch_razm_0
Sym-names.S:14               .data:00000004 Bait
Sym-names.S:18               .data:00000008 Buf0
Sym-names.S:20               .data:0000000a Buf1
Sym-names.S:22               .data:0000000c Summ_buf

```

```

Sym-names.S:26      .data:00000010 Sch_razm_1
Sym-names.S:28      .data:00000014 Slowo
Sym-names.S:32      .data:00000018 Sch_razm_2
Sym-names.S:34      .data:0000001c Dvoinoe_Slovo
Sym-names.S:37      .data:00000020 Sch_razm_3
Sym-names.S:40      .data:00000024 Adres_Stroki
Sym-names.S:43      .data:00000025

Adr_2_baita_Stroki
Sym-names.S:46      .data:00000024 Stroka
Sym-names.S:51      *ABS*:00000013 Str_Len
Sym-names.S:53      .data:00000036

Adr_posledn_baita_Stroki
Sym-names.S:56      .data:00000037

Adres_Stroki_v_0P
Sym-names.S:62      .data:0000003b

Adr_posledn_baita_Stroki_v_0P
Sym-names.S:65      .data:0000003f

Dlina_Stroki_v_0P

```

NO UNDEFINED SYMBOLS

Таблица символов исполняемого файла Sym-names

```
ybgv@ybgv-home:~/GnuAS/Labs/Sym-names.25> objdump -t Sym-names
```

```
Sym-names:      формат файла elf32-i386
```

```
SYMBOL TABLE:
```

```

00000000 1      df *ABS* 00000000 Sym-names.o
00000032 1      *ABS* 00000000 buf_Length0
08049098 1      0  .bss 00000001 c
080490a0 1      0  .bss 00000032 buf

```

```

00000050 l      *ABS* 00000000 buf_Length1
08049054 l      .data00000000 Sch_razm_0
08049058 l      .data00000000 Bait
0804905c l      .data00000000 Buf0
0804905e l      .data00000000 Buf1
08049060 l      .data00000000 Summ_buf
08049064 l      .data00000000 Sch_razm_1
08049068 l      .data00000000 Slowo
0804906c l      .data00000000 Sch_razm_2
08049070 l      .data00000000 Dvoinoe_Slovo
08049074 l      .data00000000 Sch_razm_3
08049078 l      .data00000000 Adres_Stroki
08049079 l      .data00000000 Adr_2_baita_Stroki
08049078 l      .data00000000 Stroka
00000013 l      *ABS* 00000000 Str_Len
0804908a l      .data00000000 Adr_posledn_baita_Stroki
0804908b l      .data00000000 Adres_Stroki_v_OP
0804908f l      .data00000000 Adr_posledn_baita_Stroki_v_OP
08049093 l      .data00000000 Dlina_Stroki_v_OP
00000000      *UND* 00000000 _start
08049098 g      .bss 00000000 __bss_start
08049097 g      .data00000000 _edata
080490d4 g      .bss 00000000 _end

```

Вывод таблицы символов по команде nm

Команда nm <флаги> <имя_ELF_файла> дает еще один формат

вывода.

```
ybgv@ybgv-home:~/GnuAS/Labs/Sym-names.25> nm Sym-names
```

```

08049079 d Adr_2_baita_Stroki
08049078 d Adres_Stroki
0804908b d Adres_Stroki_v_OP
0804908a d Adr_posledn_baita_Stroki
0804908f d Adr_posledn_baita_Stroki_v_OP
08049058 d Bait
08049098 B __bss_start
080490a0 b buf
0804905c d Buf0
0804905e d Buf1
00000032 a buf_Length0
00000050 a buf_Length1
08049098 b c

```

```
08049093 d Dlina_Stroki_v_OP
08049070 d Dvoinoe_Slovo
08049097 D _edata
080490d4 B _end
08049054 d Sch_razm_0
08049064 d Sch_razm_1
0804906c d Sch_razm_2
08049074 d Sch_razm_3
08049068 d Slowo
          U_start
00000013 a Str_Len
08049078 d Stroka
08049060 d Summ_buf
```

Пример.

```
ybgv@ybgv-home:~/KAPPA/.../Sym-names> nm Sym-names.o
```

```
00000025 d Adr_2_baita_Stroki
00000024 d Adres_Stroki
00000037 d Adres_Stroki_v_OP
00000036 d Adr_posledn_baita_Stroki
0000003b d Adr_posledn_baita_Stroki_v_OP
00000004 d Bait
00000008 b buf
00000008 d Buf0
0000000a d Buf1
00000032 a buf_Length0
00000050 a buf_Length1
00000000 b c
0000003f d Dlina_Stroki_v_OP
0000001c d Dvoinoe_Slovo
00000000 d Sch_razm_0
00000010 d Sch_razm_1
00000018 d Sch_razm_2
00000020 d Sch_razm_3
00000014 d Slowo
00000013 a Str_Len
00000024 d Stroka
0000000c d Summ_buf
```

Задача первой лабораторной работы

Исходный файл task1.c

/*

* Вычисление количества позиций символов для
* символьного представления в десятичной системе счисления
* неотрицательного 32-битового двоичного целого числа
*

* Программа содержательно эквивалентна программе task1.S
*

* Компиляция: gcc -m32 -ggdb -o task1-exe-c task1.c
*

* -m32 генерировать 32 разрядные машинные команды

* -ggdb - включить в исполняемый файл отладочную информацию

* -o task1-exe-c - задание имени выходного (исполняемого)
файла

* task1.c - исходный - входной файл (этот файл)
*

* Запуск отладчика: kdbg task1-exe-c
*

* Получение asm текста, сгенерированного gcc: gcc -m32 -S -o
task1-c.S task1.c
*

* -S - генерировать asm текст, не генерировать объектный и
исп. файлы

* -o task1-c.S - выходной файл программы на языке ассемблера
* в которую gcc транслирует файл task1.c

* task1.c - исходный - входной файл (этот файл)
*/

main()

{

/* соотв. конструкция ассемблера в 1.S

и коммент. */

int n = 2345; /* n: .long 2345 число */

int length = 0; /* length: .long 0 кол-во симв.

позиций */

```

/* Эти данные РАЗМЕЩАЕМ во встроенных в процессор
целых 32 разрядных переменных - регистрах с
фиксированными
именами. ПОЭТОМУ ОПИСЫВАТЬ ИХ НЕ НАДО. Нет аналога в asm
*/
int counter; /* рег. %ebx - счетчик делений */
int quotient; /* будет в рег. %eax после деления -
частное */
int remainder; /* будет в рег. %edx после деления -
остаток */

quotient = n; /* movl n, %eax */
counter = 0; /* movl $0, %ebx */

nextdigit:
remainder = quotient % 10; /* этот и следующий оператор
/* выполняются командой idivl
ten
*/
quotient = quotient / 10;

++counter; /* incl %ebx */

if (quotient)
goto nextdigit;

/* проверка условия на ЯКЦП ВСЕГДА выполняется парой команд:
cmpr{b|w|l} <операнд1>, <операнд2> - вып. вычитание
<операнд2>-<операнд1>
- отражает знак и др. свойства результата
в битах регистра флагов.

jcc <операнд> - j[ump on] c[ondition] c[ode] проверяет
условие в битах регистра флагов и по результату проверки
выполняет/не выполняет переход по адресу <операнд>.

В нашем случае команда jg nextdigit { jg - jump if greater
- перейти если строго больше} выполняет переход на метку
nextdigit: если предыдущая команда cmpl $0, %eax установила в
регистре флагов биты, указывающие, что значение в регистре
%eax,
т.е. частное, СТРОГО БОЛЬШЕ нуля.
*/
length = counter; /* movl %ebx, length */
}

```

Исходный файл task1.S

/*

* Вычисление количества позиций символов для
* символьного представления в десятичной системе счисления
* неотрицательного 32-битового двоичного целого числа

* Программа содержательно эквивалентна программе task1.c

* Ассемблирование: as -ahlsm=task1.lst --32 -gstabs+ -o
task1.o task1.S

*

* -ahlsm ключи полного листинга

* task1.lst - имя ТЕКСТОВОГО файла листинга, описывающего
* результат ассемблирования

* --32 - генерировать 32 разрядные машинные команды

* -gstabs+ - включать отладочную информацию формата stabs

* -o task1.o - задание имени выходного (для команды as -
* объектного) файла

* task1.S - исходный - входной файл (этот файл)

*

* Редактирование связей: ld -melf_i386 -o task1-exe-S
task1.o

* МОЖНО НЕ ЧИТАТЬ! В нашем курсе используется только

* -melf_i386

* -m<эмуляция ld> ld может генерировать машинный код
* исполняемого файла для нескольких архитектур.

* Говорят, что ld "эмулирует" архитектуру.

* Поддерживаемые архитектуры - "эмуляции":

* elf_x86_64

* elf_i386

* i386linux

* elf_l10m

* -o task1-exe-S - задание имени выходного

* (для команды ld - исполняемого) файла

* task1.o - объектный файл (входной для редактора связей
ld)

```
* Запуск отладчика: kdbg task1-exe-S
```

```
*
```

```
*/
```

```
.include "my-macro" # подключение файла с макроопределениями
```

```
.data # секция данных, распределение памяти  
# соотв. конструкция языка C и коммент.
```

```
n: .long 2345 # int n = 2345; число  
length: .long 0 # int length =0; результат  
ten: .long 10 # определяем константу ЯВНО  
# нет аналога в C
```

```
.text # секция команд процессора
```

```
.global _start # точка входа - глобальная метка
```

```
_start:  
nop # пустая операция - no operation
```

```
# нужна, чтобы задать отладчику контр. точку останова на  
# следующей, первой  
# содержательной команде программы
```

```
movl $0, %ebx # counter = 0; счетчик делений  
movl n, %eax # готовим к команде деления idivl  
# нет аналога в C
```

```
nextdigit:  
movl $0, %edx # еще готовим, уже в цикле  
# нет аналога в C
```

```
# Смысл этой подготовки.  
# Примем, что делим 32 битовое число, размещенное в EAX.  
# Т.к. команда idivl интерпретирует значения в паре  
регистров EDX:EAX  
# как единое 64 битовое делимое, то перед делением EDX  
должен иметь значение  
# 0. После выполнения эта команда  
# помещает в EDX - остаток, в EAX - частное, что  
# НАРУШАЕТ ИНТЕРПРЕТАЦИЮ значений пары EDX:EAX. Перед  
следующим делением эту  
# интерпретацию надо восстановить, присвоив EDX значение 0.
```

```
idivl ten # делим объединенные регистры
```

edx:eax на 10

частное в eax, остаток в edx

incl %ebx # ++counter; счетчик делений + 1

две следующие команды соответствуют условному оператору
if (quotient) goto
nextdigit;

cmpl \$0, %eax # частное > 0 ?
jg nextdigit # ДА, продолжаем

/*

проверка условия на ЯКЦП ВСЕГДА выполняется парой команд:

cmp{b|w|l} <операнд1>, <операнд2> - вып. вычитание
<операнд2>-<операнд1>

- отражает знак и др. свойства
результата
в битах регистра флагов.

jcc <операнд> - j[ump on] c[ondition] c[ode] проверяет
условие в битах регистра флагов и по результату
проверки
выполняет/не выполняет переход по адресу
<операнд>.

В нашем случае команда jg nextdigit { jg - jump if greater
- перейти если строго больше} выполняет переход на метку
nextdigit: если предыдущая команда cmpl \$0, %eax установила в
регистре флагов биты, указывающие, что значение в регистре
%eax,
т.е. частное, СТРОГО БОЛЬШЕ нуля.

*/

movl %ebx, length # length = counter; ДА, сохраняем
результат

Finish # конец работы,
возврат в ОС
(макро из файла my-macro)

.end # последняя строка исходного текста

as прекращает чтение файла исходного текста

Конец исходного файла.

Файл task1.lst 16 представление результатов ассемблирования, т.е. что получилось на ЯКЦП из task1.S.

GAS LISTING task1.S page 1

```
1      /*
2      * Вычисление количества позиций символов для
3      * символьного представления неотрицательного
4      * 32-битового целого числа
5
6      * Программа содержательно эквивалентна программе task1.c
7
8      * Ассемблирование:  as -ahlsm=task1.lst --32 -gstabs+ -o
task1.o task1.S
9      *
10     * -ahlsm ключи полного листинга
11     * task1.lst - имя ТЕКСТОВОГО файла листинга, описывающего
12     * результат ассемблирования
13
14     * --32 - генерировать 32 разрядные машинные команды
15
16     * -gstabs+ - ключи генерации отладочной информации для от
отладчиков
17
18     * -o task1.o - задание имени выходного ( для команды as -
19     * объектного) файла
20
21     * -o ключ определения имени выходного ( для команды as -
22     * объектного) файла
23
24     * task1.S - исходный - входной файл (этот файл)
25     *
26     * Редактирование связей: ld -melf_i386 -o task1-exe-S task1.o
27
28     * МОЖНО НЕ ЧИТАТЬ! В нашем курсе используется только
29     * -melf_i386
30     * -m<эмуляция ld> ld может генерировать машинный код
31     * исполняемого файла для нескольких архитектур.
32     * Говорят, что ld "эмулирует" архитектуру.
33     * Поддерживаемые архитектуры - "эмуляции":
34
35     *     elf_x86_64
36     *     elf_i386
37     *     i386linux
38     *     elf_l1om
39
40     * -o task1-exe-S - задание имени выходного
41     * (для команды ld - исполняемого) файла
```

```

42
43 * -o ключ определения имени выходного
44 * (для команды ld - исполняемого) файла
45
46 * task1.o - объектный файл (входной для редактора связей ld
47
48 * Запуск отладчика: kdbg task1-exe-S
49 *
50 */
51
; 52 .include "my-macro" # подключение файла с макроопределениями
  1 /* Макроопределение завершения работы */
  2
  3 .macro Finish
  4     movl $0, %ebx # first argument: exit code
  5     movl $1, %eax # sys_exit index

```

GAS LISTING task1.S page 2

```

  6     int    $0x80          # kernel interrupt
  7 .endm
  8
53
54
55 .data    # секция данных, распределение памяти
56         # соотв. конструкция языка C и коммент.
57 0000 29090000    n:        .long 2345    # int n = 2345; число
58 0004 00000000    length:  .long 0      # int length =0; результат
59 0008 0A000000    ten:     .long 10     # определяем константу ЯВНО
60                                     # нет аналога в C
61
62 .text # секция команд процессора
63
64 .global _start    # точка входа - глобальная метка
65
66 _start:
67 0000 90          nop # пустая операция - no operation
68
69 # нужна, чтобы задать отладчику контр. точку останова
70 # содержательной команде программы
71
72 0001 B0000000    movl $0, %ebx    # counter = 0; счетчик делений
72     00
73 0006 A1000000    movl n, %eax     # готовим к команде деления
idivl
73     00
74                                     # нет аналога в C
75 nextdigit:
76 000b BA000000    movl $0, %edx    # еще готовим, уже в
цикле
76     00
77                                     # нет аналога в C
78
79 # Смысл этой подготовки.
80 # Примем, что делим 32 битовое число, размещенное в EAX.
81 # Т.к. команда idivl интерпретирует значения в паре регистров-

```

```

СЛОВ
82 # EDX:EAX как единое 64 битовое делимое, то перед делением EDX
83 # должно быть равно 0. После выполнения деления эта команда
84 # помещает в EDX - остаток, в EAX - частное, что
85 # НАРУШАЕТ ИНТЕРПРЕТАЦИЮ значений пары EDX:EAX. Перед следующим
86 # делением эту интерпретацию надо восстановить, присвоив EDX 0
87
88 0010 F73D0800      idivl ten # делим объединенные регистры edx:eax
на 10
88      0000
89          # частное в eax, остаток в edx
90
91 0016 43          incl %ebx      # ++counter; счетчик делений + 1
92
93 # две следующие команды соответствуют условному оператору
94 # if (quotient) goto nextdigit;
95
96 0017 83F800      cmp $0, %eax   # частное > 0 ?
97 001a 7FEF        jg  nextdigit # ДА, продолжаем
98
99
100 /*
101
102 проверка условия на ЯКЦП ВСЕГДА выполняется парой команд

```

GAS LISTING task1.S page 3

```

103
104     cmp{b|w|l} <операнд1>, <операнд2> - вып. вычитание
105     - <операнд2>-<операнд1> отражает знак и др. свойства результата
106     в битах регистра флагов.
107
108     jcc <операнд> - j[ump on] c[ondition] c[ode] проверяет
109     условие в битах регистра флагов и по результату проверки
110     выполняет/не выполняет переход по адресу <операнд>
111
112     В нашем случае команда jg  nextdigit { jg - jump if greater
113     - перейти если строго больше} выполняет переход на метку
114     nextdigit: если предыдущая команда cmpl $0, %eax установила в
115     регистре флагов биты, указывающие, что значение в регистре
116     т.е. частное, СТРОГО БОЛЬШЕ нуля.
117
118     */
119
120 001c 891D0400      movl %ebx, length # length = counter;
                        НЕТ, сохраняем результат
120      0000
121
122      Finish      # конец работы,
122 0022 B8000000      > movl $0,%ebx
122      00
122 0027 B8010000      > movl $1,%eax
122      00
122 002c CD80          > int $0x80
123      # возврат в ОС
124      # (макро из файла my-macro)

```

125
126 .end # последняя строка исходного текста

GAS LISTING task1.S page 4

DEFINED SYMBOLS

task1.S:57 .data:0000000000000000 n
task1.S:58 .data:0000000000000004 length
task1.S:59 .data:0000000000000008 ten
task1.S:66 .text:0000000000000000 _start
task1.S:75 .text:000000000000000b nextdigit

NO UNDEFINED SYMBOLS

Конец файла листинга

Иллюстрация работы команд `idivl` (описание ее на стр. 101).

Делимое — 64-разрядное число в паре объединенных регистров EDX:EAX

До первого выполнения `idivl`:

EDX	EAX
0	2345

После выполнения `idivl`:

EDX	EAX
5	234
остаток	частное

NB. Записывая остаток в регистр EDX команда **ИСКАЖАЕТ** объединенное делимое.

Если перед следующим делением не присвоить регистру EDX значение 0, то при следующем делении в цикле делимое будет иметь значение 5234 и получится остаток 4, а частное 523.

Если же присвоить регистру EDX значение 0, то делимое будет иметь значение 234, остаток — 4, частное 23, что нам и нужно.

Выводы

- Регистры процессора в архитектуре IA-32 это быстродействующие 32-битные устройства памяти, имеющие символьные имена.
- as поддерживает синтаксисы AT&T и Intel.
- Символьные имена :
 - имеют двоичное значение (как правило 32-битовое) и тип – t, d, b или a (возможны также T, D, B, A);
 - определяются как метки, если за именем без пробела следует символ «:»;
 - значениями меток являются адреса данных и передач управления в ОП, формируемые в процессе ассемблирования;
 - значением метки в секции , в том числе в секциях .text, .data, .bss объектного файла, является количество байтов ОП от начала секции до адреса, соответствующего метке. Для этого значения можно использовать более общий термин, применяемый Intel – перемещение (offset).
 - метки используются для задания операндов команд, при формировании соответствующей команды ЯКЦП в процессорном формате значение метки помещается в поле «Смещение»;
 - значения меток называются перемещаемыми (relocatable), т. е. перемещаются (настраиваются) редактором связей ld для выполнения в конкретном блоке ОП, выделяемым

загрузчиком;

→ если символьные имена не заканчиваются символом «:», то они могут иметь произвольные значения разных типов, в том числе неперемещаемого редактором связей типа «постоянный» (absolute).