

# 14. Введение в программирование на языке ассемблера. Часть 3

## Разделы:

- Средства ассемблирования
- Макросредства и макропроцессор
- Встроенный ассемблер



# Псевдокоманды

- Синтаксис:

- `.equ symbol, expression`

- Псевдокоманда связывает имя с заданным выражением

- `.equ four, 4`

- Теперь можно использовать символьное имя, например:

- `movl $four, %ecx`



# Псевдокоманды

```
hello_str:                /* наша строка                */  
    .string "Hello, goodbye and farewell!\n"
```

```
                                /* длина строки                */  
    .set hello_str_length, . - hello_str - 1
```

```
.rept count
```

- **Пример кода:**

```
.rept    3  
.long    0  
.endr
```

```
.include "file"  
.incbin "file"[,skip[,count]]
```

# Вычисление выражений во время ассемблирования

- Выражение, вычисляемое ассемблером, как правило, должно быть целочисленным, т.е. состоять из констант, меток и подвыражений в скобках
- Практически все операции те же, что и в языке C, с сохранением уровней приоритетов и ассоциативностей
- Объявление:

```
.equ num_expression, (6 - 3) * (10 / 2)
```

- Каков будет результат инструкции?

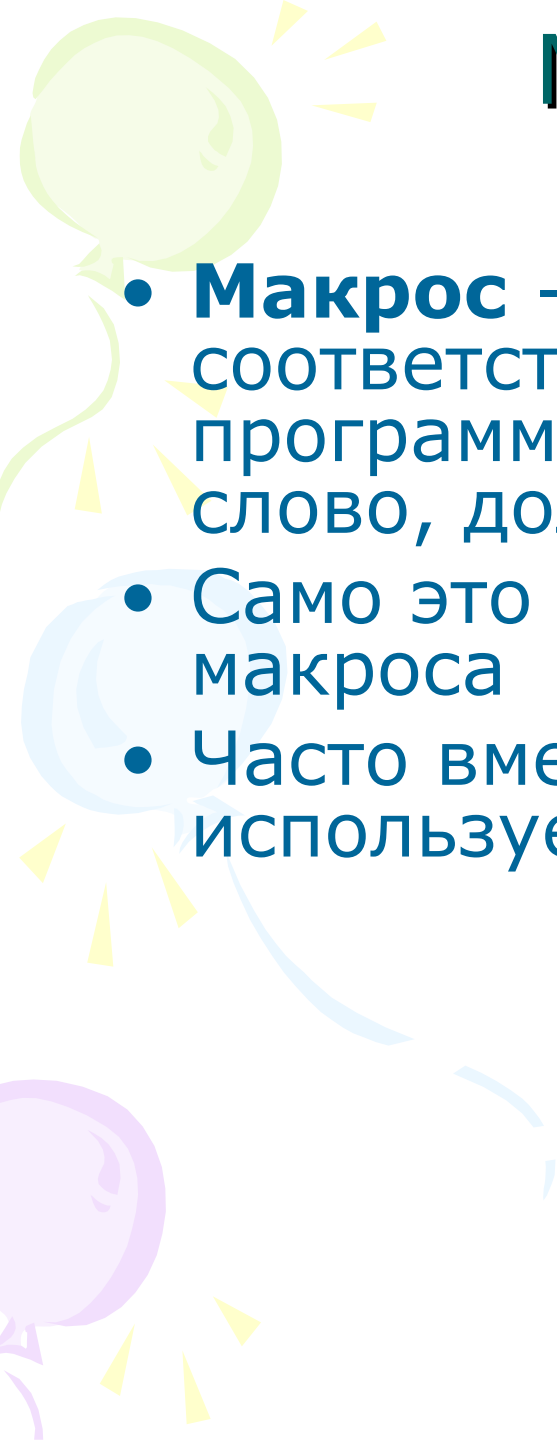
```
movl $num_expression, %ecx
```

# Макросредства и макропроцессор

- **Макропроцессор** - программное средство, которое получает на вход некоторый текст и, пользуясь указаниями из текста, частично преобразуют его, давая на выходе, в свою очередь, текст, но уже не имеющий указаний к преобразованию
- В применении к ЯП макропроцессор - это преобразователь исходного текста программы, совмещенный с компилятором: результатом является текст на языке программирования, который уже потом обрабатывается компилятором

# Макросредства и макропроцессор





# Макросредства и макропроцессор

- **Макрос** - некоторое правило, в соответствии с которым фрагмент программы, содержащий определенное слово, должен быть преобразован
- Само это слово называется **именем** макроса
- Часто вместо «имени макроса» используется слово «макрос»

# Макросредства и макропроцессор

- Бывает, что макропроцессор производит преобразование текста программы, не видя ни одного имени макроса, но повинуюсь еще более прямым указаниями в виде **макродиректив**
- Одна из макродиректив - *.include*
- Она приказывает макропроцессору заменить ее саму на содержимое файла, указанного параметром директивы

```
.include "abracadabra.inc"
```





# Однострочные макросы

```
.equ five, 5  
.set five_, 5  
_five_ = 5
```

- 
- К сожалению, отменить эти макроопределения легким образом не получится, можно лишь их переопределить



# Условная компиляция

```
.set DEBUG 1
/* Какой-то код */
#ifdef DEBUG
.string
/* здесь должен быть код печати отладочных сообщений
   */
#endif
/* Подозрительный фрагмент */
#ifdef DEBUG
.string
/* здесь должен быть код печати отладочных сообщений
   */
#endif
```

# Условная компиляция

```
.ifndef Символ
.ifdef Символ
.ifnotdef Символ /* Ассемблирует, если символ не определен */

.ifb Текст /*ассемблирует, если операнд пустой */
.ifnb Текст /*ассемблирует, если операнд непустой */

.ifc Строка1, Строка2 /* ассемблирует, если операнды идентичные. Могут использоваться или не
использоваться одинарные кавычки */
.ifnc Строка1, Строка2 /* ассемблирует, если операнды неидентичные. Могут использоваться или не
использоваться одинарные кавычки */

.ifeqs Строка1, Строка2 /* ассемблирует, если операнды идентичные. Должны использоваться двойные
кавычки */
.ifnes Строка1, Строка2 /* ассемблирует, если операнды неидентичные. Должны использоваться двойные
кавычки */

.ifeq Выражение /* Ассемблирует, если аргумент равен 0 */
.ifge Выражение /* Ассемблирует, если аргумент >= 0 */
.ifle Выражение /* Ассемблирует, если аргумент <= 0 */
.ifgt Выражение /* Ассемблирует, если аргумент > 0 */
.iflt Выражение /* Ассемблирует, если аргумент < 0 */

.ifne Выражение /* Ассемблирует, если аргумент не равен 0.
То же, что if */

.elseif /* Ассемблирует, если .if или предыдущие .elseif оказались ложными */
.else /* Аналогично .elseif, но может быть ровно одна такая ветвь кода */
```

# Многострочные макросы

```
.macro    sum from=0, to=5
.long    \from
.if      \to-\from
sum      "(\from+1)", \to
.endif
.endm
```

- Какой вызов макроса приведет к такому результату?

```
.long    0
.long    1
.long    2
.long    3
.long    4
```

# Многострочные макросы

- Синтаксис:

`.macro macname`

`.macro macname macargs ...`

- Мы можем квалифицировать аргументы макроса, чтобы показать, чтобы при всех макровыводах нужно передавать им аргументы с непустыми значениями, используя конструкцию ``:req'`
- Также можно указать, что макросу передается переменное число аргументов через конструкцию ``:vararg'`
- Мы можем задать значение по умолчанию для любого аргумента макроса с помощью конструкции ``=deflt'`
- Мы не можем объявить два макроса с одинаковыми именами, пока имя не станет объектом директивы `.purgem`

# Многострочные макросы

```
.macro comm /* макрос без аргументов */
```

```
.macro plus1 p, p1
```

```
.macro plus1 p p1
```

```
----
```

```
.macro reserve_str p1=0 p2
```

```
...
```

```
reserve_str A, B
```

```
reserve_str , B
```

```
----
```

```
.macro m p1:req, p2=0, p3:vararg
```

```
---
```

- Когда мы вызываем макрос, можно задать значение аргумента как позицией, так и ключевым словом
- *sum 7, 17* эквивалентно *sum to=17, from=7*

# Многострочные макросы

- Не всегда будет срабатывать: макрос /

```
.macro label 1
```

```
\1:
```

```
.endm
```

- Вызов '*label kuku*' не создаст в коде метку *kuku*
- Он просто в результирующий ассемблерный код вставит метку *\1*

# Многострочные макросы

- Не всегда будет срабатывать макрос *opcode*

```
.macro opcode base length
```

```
\base.\length
```

```
.endm
```

- Вызов '*opcode store l*' не создаст инструкцию *store.l*
- Вместо этого будет получено сообщение об ошибке при ассемблировании конструкции *\base.\length*



# Многострочные макросы

- Решение проблем:

- Вставка пробелов

```
.macro label 1
```

```
\1 :
```

```
.endm
```

- Использование конструкции '\()'

```
.macro opcode base length
```

```
\base\() .\length
```

```
.endm
```

- Использование альтернативного синтаксиса

```
.altmacro
```

```
.macro label 1
```

```
l&:
```

```
.endm
```

# Многострочные макросы

- Директива *.endm* завершает макроопределение
- В некоторых случаях требуется досрочное завершение макроса
- Для этого используется директива *.exitm*
- *gas* хранит число выполненных макросов в псевдо-переменной
- Мы можем вывести это число при помощи `\@`, но только внутри определения макроса

# Многострочные макросы

```
.section .data
```

```
prompt_str:
```

```
    .ascii "Enter Your Name: "
```

```
pstr_end:
```

```
    .set STR_SIZE, pstr_end - prompt_str
```

```
greet_str:
```

```
    .ascii "Hello "
```

```
gstr_end:
```

```
    .set GSTR_SIZE, gstr_end - greet_str
```

```
.section .bss
```

```
// Резервирование 32 байт памяти
```

```
    .lcomm buff, 32
```

# Многострочные макросы

```
// Макрос с двумя параметрами
// Реализует системный вызов для записи
.macro write str, str_size
    movl    $4, %eax
    movl    $1, %ebx
    movl    \str, %ecx
    movl    \str_size, %edx
    int     $0x80
.endm

// Реализует системный вызов для чтения
.macro read buff, buff_size
    movl    $3, %eax
    movl    $0, %ebx
    movl    \buff, %ecx
    movl    \buff_size, %edx
    int     $0x80
.endm
```

# Многострочные макросы

```
.section .text

.globl _start

_start:
    write $prompt_str, $STR_SIZE
    read $buff, $32

    // Чтение возвращает длину в eax
    pushl %eax

    // Вывод приветствия
    write $greet_str, $GSTR_SIZE

    popl %edx

    // edx = длина, возвращенная чтением
    write $buff, %edx

_exit:
    movl $1, %eax
    movl $0, %ebx
    int $0x80
```

# Многострочные макросы

- *.lcomm* используется для определения байтового пространства
- Название переменной следует за ключевым словом *.lcomm*
- После этого через запятую указывается величина резервируемого пространства
- Синтаксис:

```
.lcomm varname, size
```

# Встроенный ассемблер

```
asm("fsin" : "=t" (answer) : "0" (angle));
```

```
answer = sin(angle);
```

```
/usr/include/asm/io.h
```

```
/usr/src/linux/arch/i386/kernel/process.s
```

```
asm (ассемблерный шаблон  
    : выходные операнды           (не обязательные)  
    : входные операнды             (не обязательные)  
    : список «затираемых» регистров (не обязательный)  
);
```

```
asm("movl %%cr3, %0\n" : "=r"(cr3val));
```

```
A    %eax
```

```
b    %ebx
```

```
c    %ecx
```

```
d    %edx
```

```
S    %esi
```

```
D    %edi
```

# Встроенный ассемблер

```
asm("sidt %0\n" : : "m"(loc));
asm("incl %0" : "=a"(var) : "0"(var));
-----
int main(void)
{
    int x = 10, y;
    asm ("movl %1, %%eax;
        "movl %%eax, %0;"
        : "=r"(y)          /*y - выходной операнд*/
        : "r"(x)            /*x - входной операнд*/
        : "%eax");          /*%eax - «затираемый» регистр*/
}
-----
pushl %ebp
    movl %esp,%ebp
    subl $8,%esp
    movl $10,-4(%ebp)
    movl -4(%ebp),%edx      /*значение x=10 сохранено в регистре %edx*/
#ifdef APP
    /*начало конструкции asm*/
    movl %edx, %eax         /*значение x переслано в регистр %eax*/
    movl %eax, %edx         /*значение y помещено в регистр %edx*/
#endif
#ifdef NO_APP
    /*конец конструкции asm*/
    movl %edx,-8(%ebp)      /*значение y в стеке заменено значением в регистре %edx*/

```



# Встроенный ассемблер

```
int main(void)
{
    int x = 10, y;
    asm ("movl %1, %%eax;
        "movl %%eax, %0;"
        : "=&r" (y)      /* y - выходной операнд, отметим использование
                           модификатора ограничения & */
        : "r" (x)         /* x - входной операнд */
        : "%eax");        /* %eax - «затираемый» регистр */
}

-----
main:
    pushl %ebp
    movl %esp, %ebp
    subl $8, %esp
    movl $10, -4(%ebp)
    movl -4(%ebp), %ecx    /* x, входной операнд находится в регистре %ecx */
#APP
    movl %ecx, %eax
    movl %eax, %edx        /* y, выходной операнд находится в регистре %edx */
#NO_APP
    movl %edx, -8(%ebp)
```

# Встроенный ассемблер

```
asm ("cpuid"
    : "=a" (_eax),
      "=b" (_ebx),
      "=c" (_ecx),
      "=d" (_edx)
    : "a" (op));
```

---

```
movl -20(%ebp),%eax      /*сохраняем переменную 'op' в %eax - входной операнд*/
#APP
cpuid
#NO_APP
movl %eax,-4(%ebp)       /*сохраняем %eax в переменную _eax - выходной операнд*/
movl %ebx,-8(%ebp)       /*сохраняем остальные регистры в
movl %ecx,-12(%ebp)      соответствующих выходных переменных*/
movl %edx,-16(%ebp)
```

# Встроенный ассемблер

```
asm ("cld\n
    rep\n
    movsb"
    : /*входные операнды отсутствуют*/
    : "S"(src), "D"(dst), "c"(count));
```

```
-----

#define rdtsc11(val) \
    __asm__ __volatile__ ("rdtsc" : "=A" (val))
```

```
-----

#ifdef APP
    rdtsc
#else
    #NO_APP
    movl %eax, -8(%ebp)
    movl %edx, -4(%ebp)
    /*За счет ограничения А
    регистры %eax и %edx служат в качестве выходных
    операндов*/
```

# Встроенный ассемблер

```
#define __syscall4(type,name,type1,arg1,type2,arg2,type3,arg3,type4,arg4) \  
type name (type1 arg1, type2 arg2, type3 arg3, type4 arg4) \  
{ \  
    long __res; \  
    __asm__ volatile ("int $0x80" \  
        : "=a" (__res) \  
        : "0" (__NR_##name), "b" ((long) (arg1)), "c" ((long) (arg2)), \  
          "d" ((long) (arg3)), "S" ((long) (arg4))); \  
    __syscall_return(type, __res); \  
}  
-----  
__asm__ __volatile__(  
    "lock; decl %0"  
    : "=m" (counter)  
    : "m" (counter));  
-----  
#APP  
    lock  
    decl -24(%ebp) /*переменная counter модифицирована в своей локации памяти*/  
#NO_APP.
```



# Встроенный ассемблер

```
asm ("movl $count, %%ecx;  
    up: lodsl;  
    stosl;  
    loop up;"  
    :                                     /*выходные операнды отсутствуют*/  
    : "S"(src), "D"(dst)                /*входные операнды*/  
    : "%ecx", "%eax" );                 /*список «затираемых» регистров*/
```

# Встроенный ассемблер

```
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char* argv[])
{
    long max = atoi (argv[1]);
    long number;
    long i;
    unsigned position;
    volatile unsigned result;

    /* Repeat the operation for a large number of values. */
    for (number = 1; number <= max; ++number) {
        /* Repeatedly shift the number to the right, until the result is
           zero.  Keep count of the number of shifts this requires. */
        for (i = (number >> 1), position = 0; i != 0; ++position)
            i >>= 1;
        /* The position of the most significant set bit is the number of
           shifts we needed after the first one. */
        result = position;
    }

    return 0;
}
```

# Встроенный ассемблер

```
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char* argv[])
{
    long max = atoi (argv[1]);
    long number;
    unsigned position;
    volatile unsigned result;

    /* Repeat the operation for a large number of values. */
    for (number = 1; number <= max; ++number) {
        /* Compute the position of the most significant set bit using the
           bsr assembly instruction. */
        asm ("bsrl %1, %0" : "=r" (position) : "r" (number));
        result = position;
    }

    return 0;
}
```

## *See also*

- Магда, Ю.С. Ассемблер для процессоров Intel Pentium/ Ю.С. Магда. – СПб.: Питер, 2006. – 416 с.
- Робачевский, А. Операционная система Unix, 2 изд./ А.Робачевский, С.Немнюгин, О.Стесик. – СПб.: БХВ-Петербург, 2010. – 656 с.
- Столяров, А.В. Программирование на языке ассемблера NASM для ОС UNIX: учеб.пособие. – М.: Макс, 2011. – 188 с. – Доступ: [http://www.stolyarov.info/books/asm\\_unix](http://www.stolyarov.info/books/asm_unix)



## *See also*

- Использование GNU ассемблера as - <http://www.opennet.ru/docs/RUS/gas/gas.html>
- Using as - <http://sourceware.org/binutils/docs/as/index.html>
- Ассемблеры для Linux: Сравнение GAS и NASM - <http://www.ibm.com/developerworks/ru/library/l-gas-nasm/>
- Ассемблер в Linux для программистов C – [http://ru.wikibooks.org/wiki/Ассемблер\\_в\\_Linux\\_для\\_программистов\\_C](http://ru.wikibooks.org/wiki/Ассемблер_в_Linux_для_программистов_C)