

13. Введение в программирование на языке ассемблера. Часть 2

Разделы:

- Система команд x86 с примерами использования
- Конвенции вызовов подпрограмм и рекурсии
- «Плавающая» арифметика
- Системные вызовы
- Работа со структурами и файлами

Команды циклов

- Команда *loop*

- Синтаксис:

`loop метка`

- Принцип работы:

- уменьшить значение регистра `%ecx` на 1;
- если `%ecx == 0`, передать управление следующей за `loop` команде;
- если `%ecx <> 0`, передать управление на метку.

Команды циклов

```
.data
printf_format:
    .string "%d\n"

.text
.globl main
main:
    movl    $0, %eax                /* в %eax будет результат, поэтому в
                                   начале его нужно обнулить */
    movl    $100, %ecx              /* 100 шагов цикла */

sum:
    addl    %ecx, %eax              /* %eax = %eax + %ecx */
    loop    sum

    /* %eax = 5050, %ecx = 0 */

/*
 * следующий код выводит число в %eax на экран и завершает программу
 */
    pushl   %eax
    pushl   $printf_format
    call    printf
    addl    $8, %esp

    movl    $0, %eax
    ret
```

Команды циклов

```
.data
printf_format:
    .string "%d\n"
```

```
array:
    .long 100, 15, 148, 12, 151, 3, 72
array_end:
```

```
.text
.globl main
main:
```

```
    movl    $0, %eax
```

```
/* в %eax будет храниться результат;
   в начале наибольшее значение - 0 */
```

```
    movl    $array, %ebx
```

```
/* в %ebx находится адрес текущего
   элемента массива */
```

Команды циклов

```
loop_start:                                /* начало цикла */
    cmpl    %eax, (%ebx)                   /* сравнить текущий элемент массива с
                                           текущим наибольшим значением из %eax
                                           */
    jbe     less                           /* если текущий элемент массива меньше
                                           или равен наибольшему, пропустить
                                           следующий код */
    movl    (%ebx), %eax                   /* а вот если элемент массива
                                           превосходит наибольший, значит, его
                                           значение и есть новый максимум */

less:
    addl    $4, %ebx                       /* увеличить %ebx на размер одного
                                           элемента массива, 4 байта */
    cmpl    $array_end, %ebx              /* сравнить адрес текущего элемента и
                                           адрес конца массива */
    je      loop_end                       /* если они равны, выйти из цикла */
    jmp     loop_start                     /* иначе повторить цикл снова */

loop_end:

/*
 * следующий код выводит число из %eax на экран и завершает программу
 */
    pushl   %eax
    pushl   $printf_format
    call    printf
    addl    $8, %esp

    movl    $0, %eax
    ret
```

Команды циклов

```
#include <stdio.h>

int main()
{
    int array[] = { 100, 15, 148, 12, 151, 3, 72 };
    int* array_end = &array[sizeof(array) / sizeof(int)];
    int max = 0;
    int *p = array;

    do
    {
        if(*p > max)
        {
            max = *p;
        }
    } while(++p != array_end);

    printf("%d\n", max);
    return 0;
}
```

Команды циклов

```
.data
printf_format:
    .string "%d\n"
```

```
array:
    .long 100, 15, 148, 12, 151, 3, 72
```

```
array_end:
```

```
array_size:
    .long (array_end - array)/4 /* количество элементов массива */
```

```
.text
.globl main
main:
```

```
    movl $0, %eax          /* в %eax будет храниться результат;
                           в начале наибольшее значение - 0 */
    movl $0, %ecx          /* начать просмотр с нулевого элемента
                           */
```

```
loop_start:                /* начало цикла */
    cmpl %eax, array(,%ecx,4) /* сравнить текущий элемент
                           массива с текущим наибольшим
                           значением из %eax */
    jbe less                /* если текущий элемент массива меньше
                           или равен наибольшему, пропустить
                           следующий код */
    movl array(,%ecx,4), %eax /* а вот если элемент массива
                           превосходит наибольший, значит, его
                           значение и есть новый максимум */
```

Команды циклов

less:

```
    incl    %ecx          /* увеличить на 1 номер текущего
                           элемента */
    cmpl    array_size, %ecx /* сравнить номер текущего элемента с
                              общим числом элементов */
    je      loop_end      /* если они равны, выйти из цикла */
    jmp     loop_start    /* иначе повторить цикл снова */
```

loop_end:

```
/*
 * следующий код выводит число в %eax на экран и завершает программу
 */
    pushl   %eax
    pushl   $printf_format
    call    printf
    addl    $8, %esp

    movl    $0, %eax
    ret
```


Команды циклов

```
loop_start:                                /* начало цикла */

/* вот тут находится тело цикла */

cmpl    ...                               /* что-то с чем-то сравнить для
                                        принятия решения о выходе из цикла */
je      loop_end                          /* подобрать соответствующую команду
                                        условного перехода для выхода из
                                        цикла */
jmp     loop_start                        /* иначе повторить цикл снова */
loop_end:
```



Команды циклов

```
loop_start:          /* начало цикла */
    cml  ...          /* что-то с чем-то сравнить для
                        принятия решения о выходе из цикла */
    je    loop_end     /* подобрать соответствующую команду
                        условного перехода для выхода из
                        цикла */

    /* вот тут находится тело цикла */

    jmp    loop_start   /* перейти к проверке условия цикла */
loop_end:
```

Команды циклов

- Цикл:

```
for(init; cond; incr)
{
    body;
}
```

- ЭКВИВАЛЕНТЕН:

```
init;
while(cond)
{
    body;
    incr;
}
```



Логическая побитовая арифметика

- *and* источник, приёмник
- *or* источник, приёмник
- *xor* источник, приёмник
- *not* операнд
- *test* операнд_1, операнд_2
- Сдвиги см. далее

Логическая побитовая арифметика

```
testb $0b00001000, %al  /* установлен ли 3-й (с нуля) бит?  */  
je     not_set  
/* нужные биты установлены */  
not_set:  
/* биты не установлены */
```

```
testl %eax, %eax  
je     is_zero  
/* %eax != 0 */  
is_zero:  
/* %eax == 0 */
```



Логическая побитовая арифметика

```
/* Shift Arithmetic Left/SHift logical Left */  
sal/shl количество_сдвигов, назначение
```

```
/* SHift logical Right */  
shr      количество_сдвигов, назначение
```

```
/* Shift Arithmetic Right */  
sar      количество_сдвигов, назначение
```

Логическая побитовая арифметика

До сдвига:

| | |
|---------|----------------------------------|
| +----+ | +-----+ |
| ? | 10001000100010001000100010001011 |
| +----+ | +-----+ |
| Флаг CF | Операнд |

Сдвиг влево на 1 бит:

| | |
|---------|--|
| +----+ | +-----+ |
| 1 <-- | 00010001000100010001000100010110 <-- 0 |
| +----+ | +-----+ |
| Флаг CF | Операнд |

Сдвиг влево на 3 бита:

| | | |
|---------------------|---------|--|
| +-----+ | +----+ | +-----+ |
| 10 | 0 <-- | 01000100010001000100010001011000 <-- 000 |
| +-----+ | +----+ | +-----+ |
| Улетели в никуда | Флаг CF | Операнд |

Логическая побитовая арифметика

До сдвига:

| | |
|----------------------------------|---------|
| +-----+ | +----+ |
| 10001000100010001000100010001011 | ? |
| +-----+ | +----+ |
| Операнд | Флаг CF |

Логический сдвиг вправо на 1 бит:

| | |
|--|---------|
| +-----+ | +----+ |
| 0 --> 01000100010001000100010001000101 --> | 1 |
| +-----+ | +----+ |
| Операнд | Флаг CF |

Логический сдвиг вправо на 3 бита:

| | | |
|--|---------|---------------------|
| +-----+ | +----+ | +-----+ |
| 000 --> 00010001000100010001000100010001 --> | 0 | 11 |
| +-----+ | +----+ | +-----+ |
| Операнд | Флаг CF | Улетели в никуда |

Логическая побитовая арифметика

До сдвига:

| | |
|----------------------------------|---------|
| +-----+ | +----+ |
| 10001000100010001000100010001011 | ? |
| +-----+ | +----+ |
| Операнд | Флаг CF |

старший бит равен 1 ==>

==> значение отрицательное ==>

==> "вдвинуть" бит 1 ---+

|

+-----+

|

V

Арифметический сдвиг вправо на 1 бит:

| | |
|--|---------|
| +-----+ | +----+ |
| 1 --> 11000100010001000100010001000101 --> 1 | |
| +-----+ | +----+ |
| Операнд | Флаг CF |

Арифметический сдвиг вправо на 3 бита:

| | | |
|---|---------|---------------------|
| +-----+ | +----+ | +-----+ |
| 111 --> 11110001000100010001000100010001 --> 0 11 | | |
| +-----+ | +----+ | +-----+ |
| Операнд | Флаг CF | Улетели в никуда |



Логическая побитовая арифметика

```
/* ROTate Right */
```

```
ror    количество_сдвигов, назначение
```

```
/* ROTate Left */
```

```
rol    количество_сдвигов, назначение
```

Логическая побитовая арифметика

До сдвига:

+---+

| ? |

+---+

Флаг CF

+-----+

| 10001000100010001000100010001011 |

+-----+

Операнд

Циклический сдвиг влево на 1 бит:

+---+

1

1

+-----+

| 1 |

<---+

| 00010001000100010001000100010111 |

---+

+---+

|

+-----+

|

Флаг CF

V

Операнд

^

|

|

+----->--->--->-----+

1

Циклический сдвиг влево на 3 бита:

+---+

0

100

+-----+

| 0 |

<---+

| 01000100010001000100010001011100 |

---+

+---+

|

+-----+

|

Флаг CF

V

Операнд

^

|

|

+----->--->--->-----+

100

Логическая побитовая арифметика

До сдвига:

| | |
|---------------------------------|---------|
| +-----+ | +----+ |
| 1000100010001000100010001001011 | ? |
| +-----+ | +----+ |
| Операнд | Флаг CF |

Циклический сдвиг вправо на 1 бит:

| | | | |
|--|-------|---|---------|
| +-----+ | 1 | 1 | +----+ |
| ---- 1100010001000100010001000100101 | ----> | | 1 |
| | | | +----+ |
| ^ Операнд | V | | Флаг CF |
| | | | |
| +-----<--<--<-----+ | | | |
| | 1 | | |

Циклический сдвиг вправо на 3 бита:

| | | | |
|-------------------------------------|-------|---|---------|
| +-----+ | 011 | 0 | +----+ |
| ---- 0111000100010001000100010001 | ----> | | 0 |
| | | | +----+ |
| ^ Операнд | V | | Флаг CF |
| | | | |
| +-----<--<--<-----+ | | | |
| | 011 | | |



Логическая побитовая арифметика

```
/* Rotate through Carry Right */
```

```
rscr    количество_сдвигов, назначение
```

```
/* Rotate through Carry Left */
```

```
rcl     количество_сдвигов, назначение
```

Логическая побитовая арифметика

До сдвига:

| | |
|---------|------------------------------|
| +----+ | +-----+ |
| X | 1000100010001000100010001011 |
| +----+ | +-----+ |
| Флаг CF | Операнд |

Циклический сдвиг влево через CF на 1 бит:

| | | | |
|-----------------------|----------|------------------------------|-------|
| X | +----+ | +-----+ | |
| +-<- | 1 <--- | 000100010001000100010001011X | ----+ |
| | +----+ | +-----+ | |
| V | Флаг CF | Операнд | ^ |
| | | | |
| +----->--->--->-----+ | | | |

Циклический сдвиг влево через CF на 3 бита:

| | | | |
|-----------------------|----------|------------------------------|-------|
| X10 | +----+ | +-----+ | |
| +-<- | 0 <--- | 0100010001000100010001011X10 | ----+ |
| | +----+ | +-----+ | |
| V | Флаг CF | Операнд | ^ |
| | | | |
| +----->--->--->-----+ | | | |

Логическая побитовая арифметика

До сдвига:

| | |
|----------------------------------|---------|
| +-----+ | +----+ |
| 10001000100010001000100010001011 | X |
| +-----+ | +----+ |
| Операнд | Флаг CF |

Циклический сдвиг вправо через CF на 1 бит:

| | | |
|--|----------|---|
| +-----+ | +----+ | X |
| +--- X1000100010001000100010001000101 ---> | 1 ->-- | + |
| | +----+ | |
| ^ Операнд | Флаг CF | V |
| | | |
| +-----<---<---<---<-----+ | | |

Циклический сдвиг вправо через CF на 3 бита:

| | | |
|--|----------|-----|
| +-----+ | +----+ | 11X |
| +--- 11X10001000100010001000100010001 ---> | 0 ->-- | + |
| | +----+ | |
| ^ Операнд | Флаг CF | V |
| | | |
| +-----<---<---<---<-----+ | | |



Логическая побитовая арифметика

```
int main()
{
    int a = 0x11223344;
    int shift_count = 8;

    a = (a << shift_count) | (a >> (32 - shift_count));

    printf("%x\n", a);
    return 0;
}
```


Основные конвенции вызовов подпрограмм

```
pushl $p3  
pushl $p2  
pushl $p1  
call proc  
addl $12, %esp
```

Рекурсии

```
.data
printf_format:
    .string "%d\n"

.text
/* int factorial(int) */
factorial:
    pushl %ebp
    movl %esp, %ebp

    /* извлечь аргумент в %eax */
    movl 8(%ebp), %eax

    /* факториал 0 равен 1 */
    cmpl $0, %eax
    jne  not_zero

    movl $1, %eax
    jmp  return

not_zero:
    /* следующие 4 строки вычисляют выражение
       %eax = factorial(%eax - 1) */
    decl %eax
    pushl %eax
    call factorial
    addl $4, %esp

    /* извлечь в %ebx аргумент и вычислить %eax = %eax * %ebx */
    movl 8(%ebp), %ebx
    mull %ebx
```

Рекурсии

```
/* результат в паре %edx:%eax, но старшие 32 бита нужно
   отбросить, так как они не помещаются в int */
return:
    movl %ebp, %esp
    popl %ebp
    ret

.globl main
main:
    pushl %ebp
    movl %esp, %ebp

    pushl $5
    call factorial

    pushl %eax
    pushl $printf_format
    call printf

/* стек можно не выравнивать, это будет сделано
   во время выполнения эпилога */

    movl $0, %eax                /* завершить программу */

    movl %ebp, %esp
    popl %ebp
    ret
```

Рекурсии

/* здесь нет рекурсии! */

factorial:

movl 4(%esp), %ecx

cmpl \$0, %ecx

jne not_zero

movl \$1, %eax

ret

not_zero:

movl \$1, %eax

loop_start:

mull %ecx

loop loop_start

ret



Полезности: обмен

- Команда *xchg* производит обмен значениями двух операндов и имеет очень простой синтаксис

xchg *операнд_1, операнд_2*

- Один из операндов или оба должны быть регистрами
- В первом случае для доступа к другому операнду используется непосредственная адресация



Полезности: обмен

```
.data
printf_format:
    .string "%d %d\n"

val1:
    .long 100

val2:
    .long 200

array:
    .rept 80
    .byte 0
    .endr
endarray:
```



Полезности: обмен

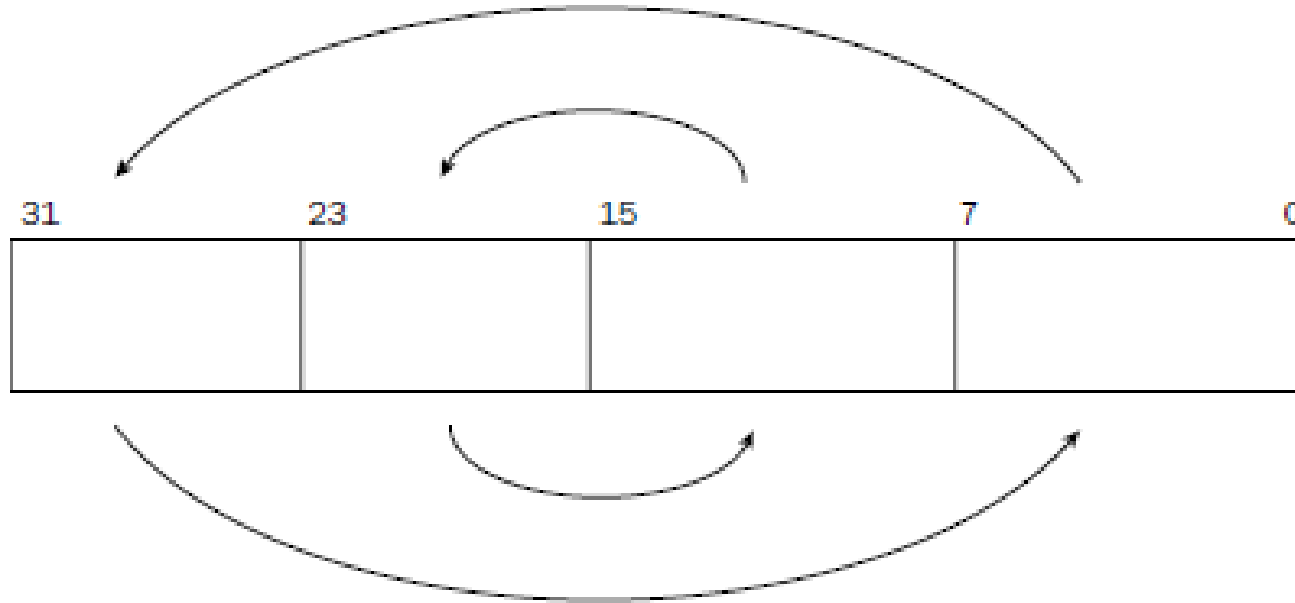
```
.text
.globl main
main:

    pushl val1
    pushl val2
    pushl $printf_format
    call  printf
    addl  $12, %esp

    movl val1, %eax
    xchg %eax, val2
    movl %eax, val1

    pushl val1
    pushl val2
    pushl $printf_format
    call  printf
    addl  $12, %esp
    movl  $0, %eax
    ret
```

Полезности: обмен



- Команда *bswap* позволяет изменять порядок байтов в 32-битном длинном слове

```
movl $0x12345678, %ebx
```

```
bswap %ebx
```


Полезности: операции с цепочками

- В ассемблере для последовательной обработки цепочек есть специализированные команды

`lods`

`stos`

- Все цепочечные команды подразумевают, что в регистре *%esi* находится указатель на следующий необработанный элемент цепочки-источника, а в регистре *%edi* – указатель на следующий элемент цепочки-приёмника

Полезности: операции с цепочками

```
.data
printf_format:
    .string "%s\n"

str_in:
    .string "abc123()!@!777"
    .set str_in_length, .-str_in

.bss
str_out:
    .space str_in_length

.text
.globl main
main:
    pushl %ebp
    movl %esp, %ebp

    movl $str_in, %esi    /* цепочка-источник */
    movl $str_out, %edi   /* цепочка-приёмник */

    movl $str_in_length - 1, %ecx /* длина строки без нулевого
                                   байта (нулевой байт не обрабатываем)
```

Полезности: операции с цепочками

```
1:
    lodsb                /* загрузить байт из источника в %al */
    incb %al            /* произвести какую-то операцию с %al */
                        */
    stosb                /* сохранить %al в приёмнике */
    loop 1b             */

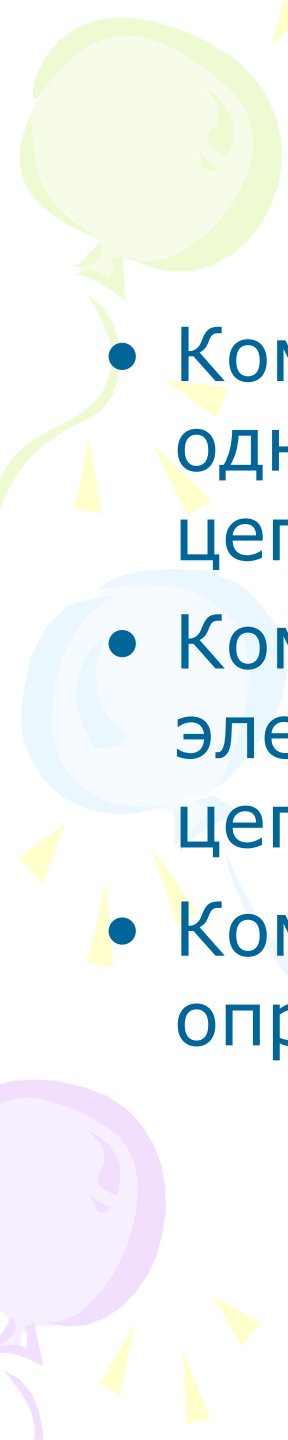
    movsb                /* копировать нулевой байт */
                        */

/* важно: сейчас %edi указывает на конец цепочки-приёмника */

    pushl $str_out
    pushl $printf_format
    call printf          /* вывести на печать */
                        */

    movl $0, %eax

    movl %ebp, %esp
    popl %ebp
    ret
```



Полезности: операции с цепочками

- Команда *movs* выполняет копирование одного элемента из цепочки-источника в цепочку-приёмник.
- Команда *cmps* выполняет сравнение элемента из цепочки-источника и цепочки-приёмника
- Команда *scas* предназначена для поиска определённого элемента в цепочке

Полезности: операции с цепочками

- Нужно организовать что-то вроде цикла для обработки всей цепочки
- Для этих целей существуют **префиксы** команд:
 - *rep*
 - *repe/repz*
 - *repne/repnz*
- Префиксы ставятся перед командой, например: *repe scas*

Полезности: операции с цепочками

- Команды для управления флагом *df*:
 - *cld* (*C*lear *D*irection *flag*) сбрасывает флаг *df*
 - *std* (*S*eT *D*irection *flag*) устанавливает флаг *df*

Полезности: операции с цепочками

```
.data
printf_format:
    .string "%s\n"

str_in:
    .string "abc123()!@!777"
    .set str_in_length, .-str_in

.bss
str_out:
    .space str_in_length

.text
/* void *my_memcpy(void *dest, const void *src, size_t n); */
my_memcpy:
    pushl %ebp
    movl %esp, %ebp
    pushl %esi
    pushl %edi
    movl 8(%ebp), %edi    /* цепочка-назначение */
    movl 12(%ebp), %esi   /* цепочка-источник */
    movl 16(%ebp), %ecx   /* длина */
    rep movsb
    movl 8(%ebp), %eax    /* вернуть dest */
    popl %edi
    popl %esi
    movl %ebp, %esp
    popl %ebp
    ret
```

Полезности: операции с цепочками

```
.globl main
main:
    pushl %ebp
    movl %esp, %ebp
    pushl $str_in_length
    pushl $str_in
    pushl $str_out
    call my_memcpy
    pushl $str_out
    pushl $printf_format
    call printf
    movl $0, %eax
    movl %ebp, %esp
    popl %ebp
    ret
```


Полезности: операции с цепочками

```
.data
printf_format:
    .string "%u\n"

str_in:
    .string "abc123()!@!777"

.text
/* size_t my_strlen(const char *s); */
my_strlen:
    pushl %ebp
    movl  %esp, %ebp
    pushl %edi
    movl  8(%ebp), %edi          /* цепочка */
    movl  $0xffffffff, %ecx
    xorl  %eax, %eax            /* %eax = 0 */
    repne scasb
    notl  %ecx
    decl  %ecx
    movl  %ecx, %eax
    popl  %edi
    movl  %ebp, %esp
    popl  %ebp
    ret
```

Полезности: операции с цепочками

```
.globl main
main:

    pushl %ebp
    movl  %esp, %ebp

    pushl $str_in
    call  my_strlen

    pushl %eax
    pushl $sprintf_format
    call  printf

    movl  $0, %eax

    movl  %ebp, %esp
    popl  %ebp
    ret
```




«Плавающая» арифметика

| FPU Register | Description |
|------------------|---|
| Data registers | Eight 80-bit registers for floating-point data |
| Status register | 16-bit register to report the status of the FPU |
| Control register | 16-bit register to control the precision of the FPU |
| Tag register | 16-bit register to describe the contents of the eight data registers |
| FIP register | 48-bit FPU instruction pointer (FIP) points to the next FPU instruction |
| FDP register | 48-bit FPU data pointer (FDP) points to the data in memory |
| Opcode register | 11-bit register to hold the last instruction processed by the FPU |



«Плавающая» арифметика

| Data Type | Length | Significand Bits | Exponent Bits | Range |
|------------------|--------|------------------|---------------|---|
| Single precision | 32 | 24 | 8 | 1.18×10^{-38} to 3.40×10^{38} |
| Double precision | 64 | 53 | 11 | 2.23×10^{-308} to 1.79×10^{308} |
| Double extended | 80 | 64 | 15 | 3.37×10^{-4932} to 1.18×10^{4932} |



«Плавающая» арифметика

```
fld/fst source /* операнд */
```

```
-----
```

```
.section .data
```

```
value1:
```

```
    .float 12.34
```

```
value2:
```

```
    .double 2353.631
```

```
.section .bss
```

```
    .lcomm data, 8
```

```
.section .text
```

```
.globl _start
```

```
_start:
```

```
    nop
```

```
    flds value1
```

```
    fldl value2
```

```
    fstl data
```

```
    movl $1, %eax
```


```
    movl $0, %ebx
```

```
    int $0x80
```



«Плавающая» арифметика

| Instruction | Description |
|-------------|---|
| FLD1 | Push +1.0 into the FPU stack |
| FLDL2T | Push $\log(\text{base } 2) 10$ onto the FPU stack |
| FLDL2E | Push $\log(\text{base } 2) e$ onto the FPU stack |
| FLDPI | Push the value of pi onto the FPU stack |
| FLDLG2 | Push $\log(\text{base } 10) 2$ onto the FPU stack |
| FLDLN2 | Push $\log(\text{base } e) 2$ onto the FPU stack |
| FLDZ | Push +0.0 onto the FPU stack |





«Плавающая» арифметика

FPU Register Stack

| | | |
|----|--|-------|
| R7 | | ST(0) |
| R6 | | ST(1) |
| R5 | | ST(2) |
| R4 | | ST(3) |
| R3 | | ST(4) |
| R2 | | ST(5) |
| R1 | | ST(6) |
| R0 | | ST(7) |

«Плавающая» арифметика

| Instruction | Description |
|-------------|------------------------------------|
| FADD | Floating-point addition |
| FDIV | Floating-point division |
| FDIVR | Reverse floating-point division |
| FMUL | Floating-point multiplication |
| FSUB | Floating-point subtraction |
| FSUBR | Reverse floating-point subtraction |

- ☐ `FADD source`: Add a 32- or 64-bit value from memory to the `ST0` register
- ☐ `FADD %st(x), %st(0)`: Add `st(x)` to `st(0)` and store the result in `st(0)`
- ☐ `FADD %st(0), %st(x)`: Add `st(0)` to `st(x)` and store the result in `st(x)`
- ☐ `FADDP %st(0), %st(x)`: Add `st(0)` to `st(x)`, store the result in `st(x)`, and pop `st(0)`
- ☐ `FADDP`: Add `st(0)` to `st(1)`, store the result in `st(1)`, and pop `st(0)`
- ☐ `FIADD source`: Add a 16- or 32-bit integer value to `st(0)` and store the result in `st(0)`



«Плавающая» арифметика

```
fadds data1 # add the 32-bit value at data1 to the ST0 register
fmull data1 # multiply the 64-bit value at data1 with the ST0 register
fidiv data1 # divide ST0 by the 32-bit integer value at data1
fsub %st, %st(1) # subtract the value in ST0 from ST1, and store in ST1
fsub %st(0), %st(1) # subtract the value in ST0 from ST1, and store in ST1
fsub %st(1), %st(0) # subtract the value in ST1 from ST0, and store in ST0
```

«Плавающая» арифметика

$$((43.65 / 22) + (76.34 * 3.1)) / ((12.43 * 6) - (140.2 / 94.21))$$

| |
|-------|
| 1 |
| 43.65 |
| |
| |
| |
| |
| |
| |
| |

| |
|---------|
| 2 |
| 1.98409 |
| |
| |
| |
| |
| |
| |
| |

| |
|---------|
| 3 |
| 76.34 |
| 1.98409 |
| |
| |
| |
| |
| |
| |

| |
|---------|
| 4 |
| 3.1 |
| 76.34 |
| 1.98409 |
| |
| |
| |
| |
| |

| |
|---------|
| 5 |
| 236.654 |
| 76.34 |
| 1.98409 |
| |
| |
| |
| |
| |

| |
|-----------|
| 6 |
| 238.63809 |
| |
| |
| |
| |
| |
| |
| |

| |
|-----------|
| 7 |
| 12.43 |
| 238.63809 |
| |
| |
| |
| |
| |
| |

| |
|-----------|
| 8 |
| 74.58 |
| 238.63809 |
| |
| |
| |
| |
| |
| |

| |
|-----------|
| 9 |
| 140.2 |
| 74.58 |
| 238.63809 |
| |
| |
| |
| |
| |

| |
|-----------|
| 10 |
| 94.21 |
| 140.2 |
| 74.58 |
| 238.63809 |
| |
| |
| |
| |

| |
|-----------|
| 11 |
| 1.48816 |
| 74.58 |
| 238.63809 |
| |
| |
| |
| |
| |

| |
|-----------|
| 12 |
| 73.09184 |
| 74.58 |
| 238.63809 |
| |
| |
| |
| |
| |

| |
|----------|
| 13 |
| 3.264907 |
| |
| |
| |
| |
| |
| |
| |



«Плавающая» арифметика

```
.section .data
value1:
    .float 43.65
value2:
    .int 22
value3:
    .float 76.34
value4:
    .float 3.1
value5:
    .float 12.43
value6:
    .int 6
value7:
    .float 140.2
value8:
    .float 94.21
output:
    .asciz "The result is %f\n"
```

«Плавающая» арифметика

```
...  
.section .text  
.globl _start  
_start:  
    finit  
    flds value1  
    fdiv value2  
    flds value3  
    flds value4  
    fmul %st(1), %st(0)  
    fadd %st(2), %st(0)  
    flds value5  
    fimul value6  
    flds value7  
    flds value8  
    fdivrp  
    fsubr %st(1), %st(0)  
    fdivr %st(2), %st(0)  
    subl $8, %esp  
    fstpl (%esp)  
    pushl $output  
    call printf  
    add $12, %esp  
    pushl $0  
    call exit
```

«Плавающая» арифметика

| Instruction | Description |
|-------------|---|
| F2XM1 | Computes 2 to the power of the value in ST0, minus 1 |
| FABS | Computes the absolute value of the value in ST0 |
| FCHS | Changes the sign of the value in ST0 |
| FCOS | Computes the cosine of the value in ST0 |
| FPATAN | Computes the partial arctangent of the value in ST0 |
| FPREM | Computes the partial remainders from dividing the value in ST0 by the value in ST1 |
| FPREM1 | Computes the IEEE partial remainders from dividing the value in ST0 by the value in ST1 |
| FPTAN | Computes the partial tangent of the value in ST0 |
| FRNDINT | Rounds the value in ST0 to the nearest integer |
| FSCALE | Computes ST0 to the ST1st power |
| FSIN | Computes the sine of the value in ST0 |
| FSINCOS | Computes both the sine and cosine of the value in ST0 |
| FSQRT | Computes the square root of the value in ST0 |
| FYL2X | Computes the value $ST1 * \log ST0$ (base 2 log) |
| FYL2XP1 | Computes the value $ST1 * \log (ST0 + 1)$ (base 2 log) |

«Плавающая» арифметика

| Instruction | Description |
|--------------|--|
| FCOM | Compare the ST0 register with the ST1 register. |
| FCOM ST(x) | Compare the ST0 register with another FPU register. |
| FCOM source | Compare the ST0 register with a 32- or 64-bit memory value. |
| FCOMP | Compare the ST0 register with the ST1 register value and pop the stack. |
| FCOMP ST(x) | Compare the ST0 register with another FPU register value and pop the stack. |
| FCOMP source | Compare the ST0 register with a 32 or 64-bit memory value and pop the stack. |
| FCOMPP | Compare the ST0 register with the ST1 register and pop the stack twice. |
| FTST | Compare the ST0 register with the value 0.0. |

| Condition | C3 | C2 | C0 |
|--------------|----|----|----|
| ST0 > source | 0 | 0 | 0 |
| ST0 < source | 0 | 0 | 1 |
| ST0 = source | 1 | 0 | 0 |



Системные вызовы

- Перечислены в файле */usr/include/asm/unistd_32.h*
- Номер задается в регистре `%eax`
- Шесть параметров, передаваемых в следующем порядке через регистры:
 - первый – в `%ebx`;
 - второй – в `%ecx`;
 - третий – в `%edx`;
 - четвертый – в `%esi`;
 - пятый – в `%edi`;
 - шестой – в `%ebp`.

Работа с файлами

- В наших программах мы работаем с файлами следующими способами:
 - Сообщаем системе имя открываемого файла и то, в каком режиме он должен быть открыт – читаем, пишем, читаем-пишем, создаем, если он не существует, и т.д.
 - Номер файлового дескриптора будет возвращен системным вызовом через регистр `%eax`
 - Далее можно оперировать с файлом, выполняя чтение и/или запись, всякий раз указывая ОС номер дескриптора
 - По окончании работы с файлами, мы должны сообщить ОС об их закрытии



Работа с файлами

```
.section .bss
```

```
.lcomm my_buffer, 500
```

```
...
```

```
movl $my_buffer, %ecx
```

```
movl 500, %edx
```

```
movl 3, %eax
```

```
int $0x80
```



Работа с файлами

- Как известно, при старте Linux-программы открыты три файла с дескрипторами:
 - *STDIN* – стандартный ввод
 - *STDOUT* – стандартный вывод
 - *STDERR* – стандартный вывод ошибок



Работа с файлами

- Программа работает с двумя файлами, читает из первого, переводит все строчные буквы в прописные и записывает результат в другой файл
- Код организован следующим образом:
 - Функция, которая принимает блок памяти и преобразует его в «верхний регистр»
 - Секция кода, которая циклически считывает данные из входного файла, размещая их в буфере, вызывает нашу функцию, и затем измененный буфер записывает в выходной файл
 - Начало функционирования программы с открытием необходимых файлов

Работа с файлами

- В нашей программе мы с помощью *.equ* свяжем имена с номерами

- Например, мы напишем

```
.equ LINUX_SYSCALL, 0x80
```

- Тогда всякий раз при использовании *LINUX_SYSCALL*, ассемблер будет подставлять вместо него 0x80

- Можно будет написать

```
int $LINUX_SYSCALL
```

- Это удобнее для чтения и запоминания

Работа с файлами

```
.section .data

/** CONSTANTS */
/* system call numbers */
.equ SYS_OPEN, 5
.equ SYS_WRITE, 4
.equ SYS_READ, 3
.equ SYS_CLOSE, 6
.equ SYS_EXIT, 1

/* options for open */
.equ O_RDONLY, 0
.equ O_CREAT_WRONLY_TRUNC, 03101

/* standard file descriptors */
.equ STDIN, 0
.equ STDOUT, 1
.equ STDERR, 2

/* system call interrupt */
.equ LINUX_SYSCALL, 0x80
.equ END_OF_FILE, 0 /* This is the return value */
                      /* of read which means we've */
                      /* hit the end of the file */
.equ NUMBER_ARGUMENTS, 2
```

Работа с файлами

```
.section .bss

/** BUFFERS
 * Buffer - this is where the data is loaded into
 * from the data file and written from
 * into the output file. This should
 * never exceed 16,000 for various
 * reasons.
 */
.equ BUFFER_SIZE, 500
.lcomm BUFFER_DATA, BUFFER_SIZE
.section .text

/* STACK POSITIONS */
.equ ST_SIZE_RESERVE, 8
.equ ST_FD_IN, -4
.equ ST_FD_OUT, -8
.equ ST_ARGC, 0      /* Number of arguments */
.equ ST_ARGV_0, 4    /* Name of program */
.equ ST_ARGV_1, 8    /* Input file name */
.equ ST_ARGV_2, 12   /* Output file name */
```

Работа с файлами

```
.globl _start  
_start:
```

```
/* save the stack pointer */  
movl %esp, %ebp
```

```
/** Allocate space for our file descriptors  
 * on the stack  
 */  
subl $ST_SIZE_RESERVE, %esp
```

```
open_files:  
open_fd_in:
```

```
/* open syscall */  
movl $SYS_OPEN, %eax  
/* input filename into %ebx */  
movl ST_ARGV_1(%ebp), %ebx  
/* read-only flag */  
movl $O_RDONLY, %ecx  
/* this doesn't really matter for reading */  
movl $0666, %edx  
/* call Linux */  
int $LINUX_SYSCALL
```

Работа с файлами

```
store_fd_in:
    /* save the given file descriptor */
    movl %eax, ST_FD_IN(%ebp)

open_fd_out:
    /* open the file */
    movl $SYS_OPEN, %eax
    /* output filename into %ebx */
    movl ST_ARGV_2(%ebp), %ebx
    /* flags for writing to the file */
    movl $O_CREAT_WRONLY_TRUNC, %ecx
    /* mode for new file (if it's created) */
    movl $0666, %edx
    /* call Linux */
    int $LINUX_SYSCALL

store_fd_out:
    /* store the file descriptor here */
    movl %eax, ST_FD_OUT(%ebp)
```


Работа с файлами

```
/** BEGIN MAIN LOOP */
read_loop_begin:
    /* READ IN A BLOCK FROM THE INPUT FILE */
    movl $SYS_READ, %eax

    /* get the input file descriptor */
    movl ST_FD_IN(%ebp), %ebx

    /* the location to read into */
    movl $BUFFER_DATA, %ecx

    /* the size of the buffer */
    movl $BUFFER_SIZE, %edx

    /* Size of buffer read is returned in %eax */
    int $LINUX_SYSCALL

    /* EXIT IF WE'VE REACHED THE END */

    /* check for end of file marker */
    cmpl $END_OF_FILE, %eax

    /* if found or on error, go to the end */
    jle end_loop
```

Работа с файлами

```
continue_read_loop:
    /** CONVERT THE BLOCK TO UPPER CASE */
    pushl $BUFFER_DATA /* location of buffer */
    pushl %eax /* size of the buffer */
    call convert_to_upper
    popl %eax /* get the size back */
    addl $4, %esp /* restore %esp */

    /** WRITE THE BLOCK OUT TO THE OUTPUT FILE */
    /* size of the buffer */
    movl %eax, %edx
    movl $SYS_WRITE, %eax

    /* file to use */
    movl ST_FD_OUT(%ebp), %ebx

    /*location of the buffer */
    movl $BUFFER_DATA, %ecx
    int $LINUX_SYSCALL

    /** CONTINUE THE LOOP */
    jmp read_loop_begin
```

Работа с файлами

```
end_loop:
    /** CLOSE THE FILES */
    movl $SYS_CLOSE, %eax
    movl ST_FD_OUT(%ebp), %ebx
    int $LINUX_SYSCALL
    movl $SYS_CLOSE, %eax
    movl ST_FD_IN(%ebp), %ebx
    int $LINUX_SYSCALL
    /** EXIT */
    movl $SYS_EXIT, %eax
    movl $0, %ebx
    int $LINUX_SYSCALL
/** This function actually does the
 * conversion to upper case for a block
 * VARIABLES:
 * %eax - beginning of buffer
 * %ebx - length of buffer
 * %edi - current buffer offset
 * %cl - current byte being examined
 * (first part of %ecx)
 *
 * CONSTANTS:
 * The lower boundary of our search
.equ LOWERCASE_A, 'a'
/* The upper boundary of our search */
.equ LOWERCASE_Z, 'z'
```

Работа с файлами

```
/* Conversion between upper and lower case */  
.equ UPPER_CONVERSION, 'A' - 'a'  
/* STACK STUFF */  
.equ ST_BUFFER_LEN, 8 /* Length of buffer */  
.equ ST_BUFFER, 12 /* actual buffer */  
convert_to_upper:  
    pushl %ebp  
    movl %esp, %ebp  
    /* SET UP VARIABLES */  
    movl ST_BUFFER(%ebp), %eax  
    movl ST_BUFFER_LEN(%ebp), %ebx  
    movl $0, %edi  
    /* if a buffer with zero length was given  
     * to us, just leave  
     */  
    cmpl $0, %ebx  
    je end_convert_loop
```

Работа с файлами

```
convert_loop:
    /* get the current byte */
    movb (%eax,%edi,1), %cl
    /* go to the next byte unless it is between */
    /* 'a' and 'z' */
    cmpb $LOWERCASE_A, %cl
    jl next_byte
    cmpb $LOWERCASE_Z, %cl
    jg next_byte
    /* otherwise convert the byte to uppercase */
    addb $UPPER_CONVERSION, %cl
    /* and store it back */
    movb %cl, (%eax,%edi,1)
next_byte:
    incl %edi #next byte
    cmpl %edi, %ebx #continue unless
    /* we've reached the end */
    jne convert_loop
end_convert_loop:
    /* no return value, just leave 8/
    movl %ebp, %esp
    popl %ebp
    ret
```

Работа со структурами данных

- Мы попробуем написать программу для манипуляции простыми структурами фиксированной длины
- Предположим, мы хотим сохранять информацию о знакомых нам людях
- Мы могли бы представить следующий пример структуры фиксированной длины:
 - Имя – 40 байт
 - Фамилия – 40 байт
 - Адрес – 240 байт
 - Возраст – 4 байта

Работа со структурами данных

```
/* record-def.s */  
.equ RECORD_FIRSTNAME, 0  
.equ RECORD_LASTNAME, 40  
.equ RECORD_ADDRESS, 80  
.equ RECORD_AGE, 320  
.equ RECORD_SIZE, 324
```

Работа со структурами данных

```
/* linux.s */  
#Common Linux Definitions  
#System Call Numbers  
.equ SYS_EXIT, 1  
.equ SYS_READ, 3  
.equ SYS_WRITE, 4  
.equ SYS_OPEN, 5  
.equ SYS_CLOSE, 6  
.equ SYS_BRK, 45  
#System Call Interrupt Number  
.equ LINUX_SYSCALL, 0x80  
#Standard File Descriptors  
.equ STDIN, 0  
.equ STDOUT, 1  
.equ STDERR, 2  
#Common Status Codes  
.equ END_OF_FILE, 0
```


Работа со структурами данных

```
/* read-record.s */  
.include "record-def.s"  
.include "linux.s"  
#PURPOSE: This function reads a record from the file  
# descriptor  
#  
#INPUT: The file descriptor and a buffer  
#  
#OUTPUT: This function writes the data to the buffer  
# and returns a status code.  
#  
#STACK LOCAL VARIABLES  
.equ ST_READ_BUFFER, 8  
.equ ST_FILEDES, 12
```

Работа со структурами данных

```
.section .text
.globl read_record
.type read_record, @function
read_record:
    pushl %ebp
    movl %esp, %ebp
    pushl %ebx
    movl ST_FILEDES(%ebp), %ebx
    movl ST_READ_BUFFER(%ebp), %ecx
    movl $RECORD_SIZE, %edx
    movl $SYS_READ, %eax
    int $LINUX_SYSCALL
    #NOTE - %eax has the return value, which we will
    # give back to our calling program
    popl %ebx
    movl %ebp, %esp
    popl %ebp
    ret
```

Работа со структурами данных

```
/* write-record.s */
#include "linux.s"
#include "record-def.s"
#PURPOSE: This function writes a record to
# the given file descriptor
#
#INPUT: The file descriptor and a buffer
#
#OUTPUT: This function produces a status code
#
#STACK LOCAL VARIABLES
.equ ST_WRITE_BUFFER, 8
.equ ST_FILEDES, 12
.section .text
.globl write_record
.type write_record, @function
```

Работа со структурами данных

```
write_record:
pushl %ebp
movl %esp, %ebp
pushl %ebx
movl $SYS_WRITE, %eax
movl ST_FILEDES(%ebp), %ebx
movl ST_WRITE_BUFFER(%ebp), %ecx
movl $RECORD_SIZE, %edx
int $LINUX_SYSCALL
#NOTE - %eax has the return value, which we will
# give back to our calling program
popl %ebx
movl %ebp, %esp
popl %ebp
ret
```

Работа со структурами данных

```
/* write-records.s */
#include "linux.s"
#include "record-def.s"
.section .data
#Constant data of the records we want to write
#Each text data item is padded to the proper
#length with null (i.e. 0) bytes.
#.rept is used to pad each item. .rept tells
#the assembler to repeat the section between
#.rept and .endr the number of times specified.
#This is used in this program to add extra null
#characters at the end of each field to fill
#it up
record1:
.ascii "Fredrick\0"
.rept 31 #Padding to 40 bytes
.byte 0
.endr
.ascii "Bartlett\0"
.rept 31 #Padding to 40 bytes
.byte 0
.endr
.ascii "4242 S Prairie\nTulsa, OK 55555\0"
.rept 209 #Padding to 240 bytes
.byte 0
.endr
.long 45
```

Работа со структурами данных

```
record2:
.ascii "Marilyn\0"
.rept 32 #Padding to 40 bytes
.byte 0
.endr
.ascii "Taylor\0"
.rept 33 #Padding to 40 bytes
.byte 0
.endr
.ascii "2224 S Johannan St\nChicago, IL 12345\0"
.rept 203 #Padding to 240 bytes
.byte 0
.endr
.long 29
record3:
.ascii "Derrick\0"
.rept 32 #Padding to 40 bytes
.byte 0
.endr
.ascii "McIntire\0"
.rept 31 #Padding to 40 bytes
.byte 0
.endr
```

Работа со структурами данных

```
.ascii "500 W Oakland\nSan Diego, CA 54321\0"
.rept 206 #Padding to 240 bytes
.byte 0
.endr
.long 36
#This is the name of the file we will write to
file_name:
.ascii "test.dat\0"
.equ ST_FILE_DESCRIPTOR, -4
.globl _start
_start:
#Copy the stack pointer to %ebp
movl %esp, %ebp
#Allocate space to hold the file descriptor
subl $4, %esp
#Open the file
movl $SYS_OPEN, %eax
movl $file_name, %ebx
movl $O_CREAT, %ecx #This says to create if it
#doesn't exist, and open for
#writing
movl $O_WRONLY, %edx
int $LINUX_SYSCALL
```

Работа со структурами данных

```
#Store the file descriptor away
movl %eax, ST_FILE_DESCRIPTOR(%ebp)
#Write the first record
pushl ST_FILE_DESCRIPTOR(%ebp)
pushl $record1
call write_record
addl $8, %esp
#Write the second record
pushl ST_FILE_DESCRIPTOR(%ebp)
pushl $record2
call write_record
addl $8, %esp
#Write the third record
pushl ST_FILE_DESCRIPTOR(%ebp)
pushl $record3
call write_record
addl $8, %esp
#Close the file descriptor
movl $SYS_CLOSE, %eax
movl ST_FILE_DESCRIPTOR(%ebp), %ebx
int $LINUX_SYSCALL
#Exit the program
movl $SYS_EXIT, %eax
movl $0, %ebx
int $LINUX_SYSCALL
```


Работа со структурами данных

```
/* count-chars.s */
#PURPOSE: Count the characters until a null byte is reached.
#INPUT: The address of the character string
#OUTPUT: Returns the count in %eax
#PROCESS:
# Registers used:
# %ecx - character count
# %al - current character
# %edx - current character address
.type count_chars, @function
.globl count_chars
#This is where our one parameter is on the stack
.equ ST_STRING_START_ADDRESS, 8
count_chars:
pushl %ebp
movl %esp, %ebp
#Counter starts at zero
movl $0, %ecx
#Starting address of data
movl ST_STRING_START_ADDRESS(%ebp), %edx
```

Работа со структурами данных

```
count_loop_begin:
#Grab the current character
movb (%edx), %al
#Is it null?
cmpb $0, %al
#If yes, we're done
je count_loop_end
#Otherwise, increment the counter and the pointer
incl %ecx
incl %edx
#Go back to the beginning of the loop
jmp count_loop_begin
count_loop_end:
#We're done. Move the count into %eax
#and return.
movl %ecx, %eax
popl %ebp
ret
```

Работа со структурами данных

```
/* write-newline.s */
#include "linux.s"
.globl write_newline
.type write_newline, @function
.section .data
newline:
.ascii "\n"
.section .text
.equ ST_FILEDES, 8
write_newline:
pushl %ebp
movl %esp, %ebp
movl $SYS_WRITE, %eax
movl ST_FILEDES(%ebp), %ebx
movl $newline, %ecx
movl $1, %edx
int $LINUX_SYSCALL
movl %ebp, %esp
popl %ebp
ret
```

Работа со структурами данных

```
/* read-records.s */
#include "linux.s"
#include "record-def.s"
.section .data
file_name:
.ascii "test.dat\0"
.section .bss
.lcomm record_buffer, RECORD_SIZE
.section .text
#Main program
.globl _start
_start:
#These are the locations on the stack where
#we will store the input and output descriptors
#(FYI - we could have used memory addresses in
#a .data section instead)
.equ ST_INPUT_DESCRIPTOR, -4
.equ ST_OUTPUT_DESCRIPTOR, -8
#Copy the stack pointer to %ebp
movl %esp, %ebp
#Allocate space to hold the file descriptors
subl $8, %esp
```

Работа со структурами данных

```
#Open the file
movl $SYS_OPEN, %eax
movl $file_name, %ebx
movl $0, %ecx #This says to open read-only
movl $0666, %edx
int $LINUX_SYSCALL
#Save file descriptor
movl %eax, ST_INPUT_DESCRIPTOR(%ebp)
#Even though it's a constant, we are
#saving the output file descriptor in
#a local variable so that if we later
#decide that it isn't always going to
#be STDOUT, we can change it easily.
movl $STDOUT, ST_OUTPUT_DESCRIPTOR(%ebp)
record_read_loop:
pushl ST_INPUT_DESCRIPTOR(%ebp)
pushl $record_buffer
call read_record
addl $8, %esp
```

Работа со структурами данных

```
#Returns the number of bytes read.
#If it isn't the same number we
#requested, then it's either an
#end-of-file, or an error, so we're
#quitting
cmpl $RECORD_SIZE, %eax
jne finished_reading
#Otherwise, print out the first name
#but first, we must know it's size
pushl $RECORD_FIRSTNAME + record_buffer
call count_chars
addl $4, %esp
movl %eax, %edx
movl ST_OUTPUT_DESCRIPTOR(%ebp), %ebx
movl $SYS_WRITE, %eax
movl $RECORD_FIRSTNAME + record_buffer, %ecx
int $LINUX_SYSCALL
pushl ST_OUTPUT_DESCRIPTOR(%ebp)
call write_newline
addl $4, %esp
jmp record_read_loop
finished_reading:
movl $SYS_EXIT, %eax
movl $0, %ebx
int $LINUX_SYSCALL
```

Работа со структурами данных

```
/* add-year.s */  
.include "linux.s"  
.include "record-def.s"  
.section .data  
input_file_name:  
.ascii "test.dat\0"  
output_file_name:  
.ascii "testout.dat\0"  
.section .bss  
.lcomm record_buffer, RECORD_SIZE  
#Stack offsets of local variables  
.equ ST_INPUT_DESCRIPTOR, -4  
.equ ST_OUTPUT_DESCRIPTOR, -8  
.section .text  
.globl _start  
_start:  
#Copy stack pointer and make room for local variables  
movl %esp, %ebp  
subl $8, %esp
```

Работа со структурами данных

```
#Open file for reading
movl $SYS_OPEN, %eax
movl $input_file_name, %ebx
movl $0, %ecx
movl $0666, %edx
int $LINUX_SYSCALL
movl %eax, ST_INPUT_DESCRIPTOR(%ebp)
#Open file for writing
movl $SYS_OPEN, %eax
movl $output_file_name, %ebx
movl $0101, %ecx
movl $0666, %edx
int $LINUX_SYSCALL
movl %eax, ST_OUTPUT_DESCRIPTOR(%ebp)
loop_begin:
pushl ST_INPUT_DESCRIPTOR(%ebp)
pushl $record_buffer
call read_record
addl $8, %esp
```


Работа со структурами данных

```
#Returns the number of bytes read.  
#If it isn't the same number we  
#requested, then it's either an  
#end-of-file, or an error, so we're  
#quitting  
cmpl $RECORD_SIZE, %eax  
jne loop_end  
#Increment the age  
incl record_buffer + RECORD_AGE  
#Write the record out  
pushl ST_OUTPUT_DESCRIPTOR(%ebp)  
pushl $record_buffer  
call write_record  
addl $8, %esp  
jmp loop_begin  
loop_end:  
movl $SYS_EXIT, %eax  
movl $0, %ebx  
int $LINUX_SYSCALL
```

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
- flat assembler - <http://www.flatassembler.net/>
- The Netwide Assembler: NASM - <http://nasm.us/>
- Intel® 64 and IA-32 Architectures Optimization Reference Manual, 3.5.1.3 Using LEA.
- Intel® 64 and IA-32 Architectures Software Developer's Manual, 4.1 Instructions (N-Z), PUSH
- Ассемблер в Linux для программистов С – http://ru.wikibooks.org/wiki/Ассемблер_в_Linux_для_програ