Министерство образования Республики Беларусь

Учреждение Образования

«Белорусский государственный университет информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

Кафедра электронных вычислительных машин

Лабораторная работа №3

по дисциплине «Конструирование программ и языки программирования»

на тему «Обработка символьных данных»

вариант №4

|  |  |  |
| --- | --- | --- |
| Выполнил  студент гр. 250502  Грибовская А.А. |  | Проверил  Туровец Н.О. |

Минск 2023

**Цель работы:** Ознакомиться с директивами определения данных, изучить команды пересылки данных и передачи управления, изучить строчные операции и прерывания консольного ввода-вывода высокого уровня.

**Теоретические сведения**

Для выполнения работы требуется рассмотреть следующие элементы

языка ассемблера и операционной системы:

1. Арифметические операции над целыми числами.

Арифметические операции над целыми числами в двоичной арифметике

выполняются с помощью следующих команд:

-- команды сложения (ADD, ADC): Команда ADD выполняет арифметическое сложение приемника и источника, помещает сумму в приемник, не изменяя содержимое источника. Команда ADD никак не различает числа со знаком и без знака, но, употребляя значения флагов CF, OF и SF, можно использовать ее и для тех, и для других. Команда ADC во всем аналогична ADD, кроме того, что она выполняет арифметическое сложение приемника, источника и флага СF. Пара команд ADD/ADC используется для сложения чисел повышенной точности.

-- команды вычитания (SUB, SBB):

Команда SUB вычитает источник из приемника и помещает разность в приемник. Точно так же, как и команда ADD, SUB не делает различий между числами со знаком и без знака, но флаги позволяют использовать ее как для тех, так и для других. Команда SBB во всем аналогична SUB, кроме того, что она вычитает из приемника значение источника и дополнительно вычитает значение флага CF.

-- команды умножения (MUL, IMUL):

Команда MUL выполняет умножение содержимого и регистра AL, АХ, ЕАХ и помещает результат в АХ, DX:AX, EDX:EAX соответственно. Команда IMUL выполняет умножение с учетом знака. Эта команда имеет три формы, различающиеся числом операндов: a) IMUL источник. b) IMUL приемник.

c) IMUL приемник, источник1, источник2: источник 1 умножается на источник 2, и результат заносится в приемник.

-- команды деления (DIV, IDIV):

Команда DIV выполняет целочисленное деление без знака AL, АХ или

ЕАХ на источник и помещает результат в AL, АХ или ЕАХ, а остаток — в АН, DX или EDX соответственно. Команда IDIV выполняет целочисленное деление со знаком AL, АХ или ЕАХ на источник и помещает результат в AL, АХ или ЕАХ, а остаток — в АН, DX или EDX соответственно. В остальном IDIV соответствует DIV.

1. Логические побитовые операции над целыми числами.

Логические побитовые операции над целыми числами выполняются с помощью следующих команд:

-- логическое И (AND):

Команда AND выполняет «логическое И» над приемником и источником и помещает результат в приемник.

-- логическое ИЛИ (OR):

Команда OR выполняет «логическое ИЛИ» над приемником и источником и помещает результат в приемник.

-- логическое исключающее ИЛИ (XOR):

Команда XOR выполняет «исключающие ИЛИ» над приемником и источником и помещает результат в приемник.

-- инверсия (NOT):

Команда NOT выполняет инверсию бит приемника (регистр или переменная). Флаги не затрагиваются.

1. Процедуры.

Процедура в ассемблере – это аналог функции C и т.п. Ассемблер не накладывает на процедуры никаких ограничений – на любой адрес программы можно передать управление командой CALL, и оно вернется к вызвавшей процедуре, как только встретится команда RET. Такая свобода выражения легко может приводить к трудночитаемым программам, и в язык ассемблера были включены директивы логического оформления процедур.

Описание операндов PROC:

-- *метка* – название процедуры.

-- *тип* может принимать значения NEAR и FAR, и если он указан, все команды RET в теле процедуры будут заменены соответственно на RETN и RETF. По умолчанию подразумевается, что процедура имеет тип NEAR в моделях памяти TINY, SMALL и COMPACT.

-- *язык* действует аналогично такому же операнду директивы .MODEL, определяя взаимодействие процедуры с языками высокого уровня. В некоторых ассемблерах директива PROC позволяет также считать параметры, передаваемые вызывающей программой. В этом случае указание языка необходимо, так как различные языки высокого уровня используют разные способы передачи параметров.

-- *USES регистры* – список регистров, значения которых изменяет процедура. Ассемблер помещает в начало процедуры набор команд PUSH, а перед командой RET – набор команд POP, так что значения перечисленных регистров будут восстановлены.

Параметры в процедуры можно передавать в регистрах, в глобальных переменных, в стеке, в потоке кода, в блоке параметров. При передаче параметров в стеке, для чтения параметров из стека в процедуре обычно используют не команду POP, а регистр ВР, в который помещают адрес вершины стека после входа в процедуру. Для удобства ссылок на параметры, переданные в стеке, внутри функции иногда используют директивы EQU, чтобы не писать каждый раз точное смещение параметра от начала активационной записи (то есть от ВР).

**Код программы (.exe)**

.model small ; модель памяти для .exe

.stack 100h ; инициализация сдвига стека

.data ; сегмент данных

new\_line db 0Ah, 0Dh, "$" ; строка перехода на новую строку

allowed\_chars\_for\_frist db "-0123456789ABCDEF" ; допустимые символы для первого символа числа

allowed\_chars db "0123456789ABCDEF" ; допустимые символы для остальных символов числа

input\_message\_1 db "input number 1: $" ; сообщение ввести первое число

input\_message\_2 db "input number 2: $" ; сообщение ввести второе число

input\_again\_message db "input again: $" ; сообщение ввести число еще раз

and\_output\_message db "AND: $" ; вывод операции AND

or\_output\_message db "OR: $" ; вывод операции OR

xor\_output\_message db "XOR: $" ; вывод операции XOR

not\_output\_message\_1 db "NOT 1: $" ; вывод операции NOT для первого числа

not\_output\_message\_2 db "NOT 2: $" ; вывод операции NOT для второго числа

string\_max\_size equ 6 ; константа, максимальный размер буфера

input\_string db string\_max\_size, ?, string\_max\_size dup("$") ; буфер для вводимой строки

.code

start proc

mov ax, @data ; загрузка сегмента данных в регистры DS ES

mov ds,ax

mov es,ax

lea dx, input\_message\_1 ; вывод сообщения ввести первое число

call print

lea dx, input\_string ; получение первого числа в буфер

lea si, input\_string + 1

call get\_user\_input

mov ax, dx ; сохранение числового значения введенной строки в AX

push ax ; сохранение AX в стеке

lea dx, input\_message\_2 ; вывод сообщения ввести второе число

call print

lea dx, input\_string ; получение второго числа в буфер

lea si, input\_string + 1

call get\_user\_input

pop ax ; возврачение AX из стека

mov bx, dx ; сохранение числового значения второго введенного числа в BX

;-- AND --

push ax

push bx

and ax, bx

lea si, input\_string

call word\_to\_hex\_string

lea dx, and\_output\_message

call print

lea dx, input\_string

call print

lea dx, new\_line

call print

pop bx

pop ax

;-- OR --

push ax

push bx

or ax, bx

lea si, input\_string

call word\_to\_hex\_string

lea dx, or\_output\_message

call print

lea dx, input\_string

call print

lea dx, new\_line

call print

pop bx

pop ax

;-- XOR --

push ax

push bx

xor ax, bx

lea si, input\_string

call word\_to\_hex\_string

lea dx, xor\_output\_message

call print

lea dx, input\_string

call print

lea dx, new\_line

call print

pop bx

pop ax

;-- NOT 1 --

push ax

push bx

not ax

lea si, input\_string

call word\_to\_hex\_string

lea dx, not\_output\_message\_1

call print

lea dx, input\_string

call print

lea dx, new\_line

call print

pop bx

pop ax

;-- NOT 2 --

push ax

push bx

not bx

lea si, input\_string

mov ax, bx ; необходимо занести в AX, т.к. процедура превращает значение AX в строку

call word\_to\_hex\_string

lea dx, not\_output\_message\_2

call print

lea dx, input\_string

call print

lea dx, new\_line

call print

pop bx

pop ax

call exit

start endp

; \*\* you have to move offset of your string into dx before using this \*\*

print proc

push ax

mov ah, 9

int 21h

pop ax

ret

print endp

; \*\* you have to move offset of your buffer into dx before using this \*\*

; \*\* yout have to move offset of your stirng's length to si before using this \*\*

get\_user\_input proc

push ax

push cx

; get user input into buff

mov ah, 0Ah

int 21h

; add space and $ at the end

xor cx, cx

mov bx, si

mov cl, [si]

inc cl

add si, cx

mov al, '$'

mov [si], al

; go to the new line in console

mov ah, 9

push dx

xor dx, dx

lea dx, new\_line

int 21h

pop dx

lea si, input\_string

call validate\_16bit\_hex\_string

lea si, input\_string

call hex\_string\_to\_word

pop cx

pop ax

ret

get\_user\_input endp

validate\_16bit\_hex\_string proc ; проверяет что строка правильное знаковое 16-битное число

push ax

push bx

push cx

push dx

push si

push di

xor ax, ax

; si указывает на весь размер буфера

inc si ; инкремент si чтобы указывало на реальный размер

xor cx, cx

mov cl, [si] ; сохранение реального размера в CX

inc si ; инкремент si чтобы указывало на первый элемент

lodsb ; загрузка в AX первого элемента

cmp ax, '-'

je skip\_minus\_check ; в случае если число отрицательное буфер не позволит ввести более 5 символов

cmp cx, 5 ; проверка на длину строки, в случае если не отрицательное максимально 4

je validation\_failed

skip\_minus\_check:

lea di, allowed\_chars\_for\_frist

push cx

mov cx, 17 ; размер строки allowed\_chars\_for\_frist

repne scasb ; повторять пока AX не равно хотя бы одному символу из allowed\_chars\_for\_first

pop cx

jne validation\_failed ; если AX не равно ни одному из символов то это не верное число

dec cx ; декрементируем CX потому что уже проверили первый символ

validate\_loop:

lodsb ; загрузка из SI в AX

lea di, allowed\_chars

push cx

mov cx, 16

repne scasb ; повторять пока AX не равно хотя бы одному символу из allowed\_chars

pop cx

jne validation\_failed ; если AX не равно ни одному из допустимых символов это ошибка

loop validate\_loop

jmp validation\_success ; если все верно, то переход на конец процедуры

validation\_failed: ; иначе повторное прошение числа

lea dx, input\_again\_message

call print

lea dx, input\_string

lea si, input\_string + 1

call get\_user\_input

validation\_success:

pop di

pop si

pop dx

pop cx

pop bx

pop ax

ret

validate\_16bit\_hex\_string endp

;ax -> word number

;si -> pointer to buffer

;[si] <- hex string

word\_to\_hex\_string proc ; процедура переводит число в строку

push ax

push bx

push cx

push dx

; fill buffer with $

push ax

push di

mov di, si ; SI указывает на начало input\_string, заполняем DI для использования stosb

mov al, '$' ; в аккумулятор кладем знак доллара

mov cx, 6 ; всего размер буфера равен 6, нужно для цикла

fill\_buffer\_with\_dollar\_loop:

stosb ; заполняем буфер со знаком доллара

loop fill\_buffer\_with\_dollar\_loop

pop di

pop ax

mov bx, ax

shr bx, 15

cmp bx, 1

jne number\_not\_negative

neg ax

mov [si], '-'

inc si

number\_not\_negative:

mov dx, ax

and dl, 00001111b ;move bits to get small tetrade

call tetrage\_to\_char

mov [si]+3, dl

mov dx, ax

shr dl, 4 ;move bits to get big tetrade

call tetrage\_to\_char

mov [si]+2, dl

mov dx, ax

mov dl, dh

and dl, 00001111b ;move bits to get small tetrade

call tetrage\_to\_char

mov [si]+1, dl

mov dx, ax

mov dl, dh

shr dl, 4 ;move bits to get big tetrade

call tetrage\_to\_char

mov [si], dl

pop dx

pop cx

pop bx

pop ax

ret

word\_to\_hex\_string endp

;si -> pointer to buffer with string

;dx <- return word

hex\_string\_to\_word proc ; процедура переводит строку в число

push ax

push bx

push cx

xor cx, cx ;cx == 0

mov cl, [si]+1 ;string length to cx

xor dx, dx ;dx == 0

inc si ;si to string front

hex\_string\_to\_word\_loop:

inc si ;move in buffer

mov bl, [si]

cmp bl, '-'

jne non\_negative

mov bh, bl

inc si

dec cx

non\_negative:

shl dx, 4 ;move bits in dx (0000 0000 0000 0000) <-4

xor ax, ax ;ax == 0

mov al, [si] ;ax == &si

cmp al, '9' ;if (ax > '9') it is a letter ABCDEF

jg letter

sub al, '0' ;get tetrade number

jmp not\_a\_letter\_skip

letter:

and al, 11011111b ;uppercase

sub al,'A'-10 ;get tetrade number

not\_a\_letter\_skip:

or dx, ax ;push into dx

loop hex\_string\_to\_word\_loop

cmp dx, 8000h

jne not\_8000h

cmp bh, '-'

je not\_overflow

jne this\_is\_overflow

not\_8000h:

mov cx, dx ; DX число после считывания строки

shr cx, 15 ; чтобы не было переполнения оно не должно быть сейчас в доп коде

cmp cx, 1

jne not\_overflow

this\_is\_overflow: ; это переполнение, просим ввести еще раз

lea dx, input\_again\_message

call print

lea dx, input\_string

lea si, input\_string + 1

call get\_user\_input

not\_overflow:

cmp bh, '-' ; если в начале строки знак минуса то нужно перевести в доп код

jne exit\_conversion

neg dx ; перевод в доп код

exit\_conversion:

pop cx

pop bx

pop ax

ret

hex\_string\_to\_word endp

;dl -> register with tetrade

;dl <- ascii of char

tetrage\_to\_char proc

add dl, '0' ;add ascii of '0' to get dec num char

cmp dl, '9' ;if al > '9' it is a letter

jle tetrage\_to\_char\_end

add dl, 7 ;if letter get right letter by adding 7

tetrage\_to\_char\_end:

ret

tetrage\_to\_char endp

exit proc

mov ah, 4ch

int 21h

exit endp

end start

**Вывод программы**

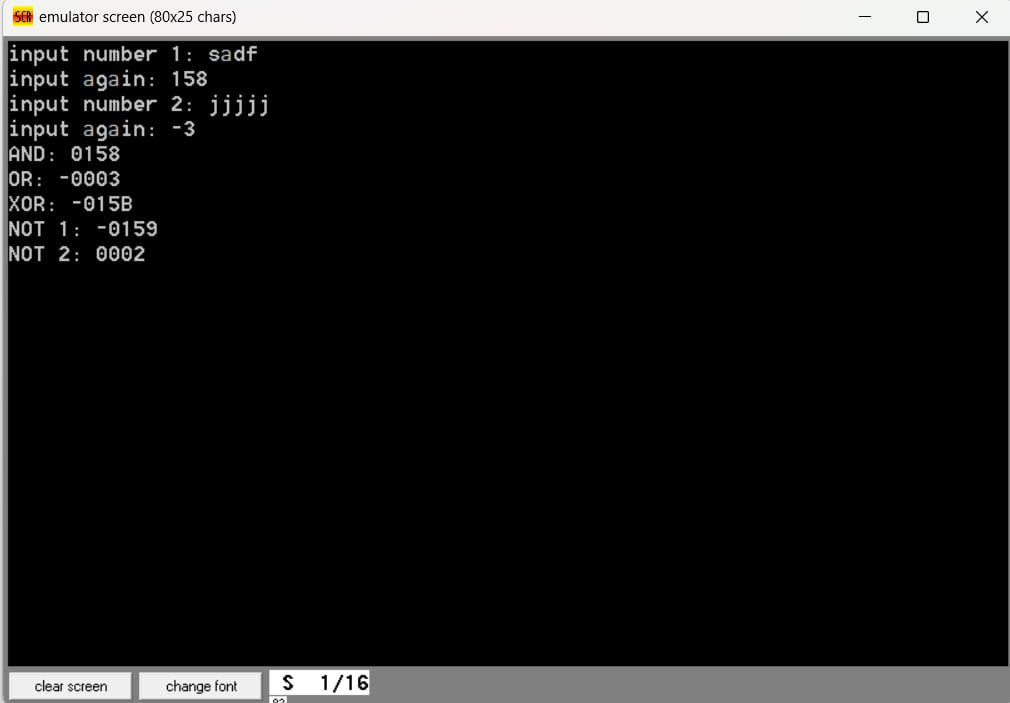


Рисунок 1 – Результат работы программы