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 «Программирование на языке ассемблера»

на тему «Создание простой программы на языке ассемблера»

вариант №7

|  |  |  |
| --- | --- | --- |
| Выполнил  студент гр. 250504  Казаченко П.Е. |  | Проверил  Туровец Н.О. |

Минск 2023

**Цель работы**: Ознакомиться с арифметическими операциями над целочислен- ными данными, обработкой массивов чисел, ознакомиться с правилами оформ- ления ассемблерных процедур.

## Теоретические сведения

Для выполнения работы требуется рассмотреть следующие элементы языка ассемблера:

* 1. *Арифметические операции над целыми числами.*

Арифметические операции над целыми числами в двоичной арифметике выполняются с помощью следующих команд:

-- команды сложения (ADD, ADC): ADD приемник, источник

Команда ADD выполняет арифметическое сложение приемника и источ- ника, помещает сумму в приемник, не изменяя содержимое источника. Прием- ник может быть регистром или переменной, источник может быть числом, ре- гистром или переменной, но нельзя использовать переменную одновременно и для источника, и для приемника. Команда ADD никак не различает числа со знаком и без знака, но, употребляя значения флагов CF (перенос при сложении чисел без знака), OF (перенос при сложении чисел со знаком) и SF (знак ре- зультата), можно использовать ее и для тех, и для других.

ADC приемник, источник

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): SUB приемник, источник

Команда SUB вычитает источник из приемника и помещает разность в приемник. Приемник может быть регистром или переменной, источник может быть числом, регистром или переменной, но нельзя использовать переменную одновременно и для источника, и для приемника. 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.

SBB приемник, источник

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): MUL источник

Команда MUL выполняет умножение содержимого источника (регистр или переменная) и регистра AL, АХ, ЕАХ (в зависимости от размера источника или оператора PTR) и помещает результат в АХ, DX:AX, EDX:EAX соответ- ственно. При умножении 8-битовых операндов результат всегда помещается в регистр AX. При умножении 16-битовых данных результат, который может быть длиною до 32 бит, помещается в пару регистров: в регистре DX содержат- ся старшие 16-бит, а в регистре AX – младшие 16-бит. Если старшая половина результата (АН, DX, EDX) содержит только нули (результат целиком поместился в младшую половину), то флаги CF и OF устанавливаются в 0, иначе – в 1. Зна- чение остальных флагов (SF, ZF, AF и PF) не определено.

IMUL ...

Команда IMUL выполняет умножение с учетом знака. Эта команда имеет три формы, различающиеся числом операндов:

1. IMUL источник: источник (регистр или переменная) умножается на AL, АХ или ЕАХ (в зависимости от размера операнда), и результат располагается в АХ, DX:AX или EDX:EAX соответственно.
2. IMUL приемник, источник: источник (число, регистр или перемен- ная) умножается на приемник (регистр), и результат заносится в приемник.
3. IMUL приемник, источник1, источник2: источник 1 (регистр или переменная) умножается на источник 2 (число), и результат заносится в прием- ник (регистр).

Во всех трех вариантах считается, что результат может занимать в два ра- за больше места, чем размер источника. В первом случае приемник автоматиче- ски оказывается достаточно большим, но во втором и третьем случаях могут произойти переполнение и потеря старших бит результата. Флаги OF и CF бу- дут равны единице, если это произошло, и нулю, если результат умножения

поместился целиком в приемник (во втором и третьем случаях) или в младшую половину приемника (в первом случае). Значения флагов SF, ZF, AF и PF после команды IMUL не определены.

-- команды деления (DIV, IDIV): DIV источник

Команда DIV выполняет целочисленное деление без знака AL, АХ или ЕАХ (в зависимости от размера источника) на источник (регистр или перемен- ная) и помещает результат в AL, АХ или ЕАХ, а остаток — в АН, DX или EDX со- ответственно. Результат всегда округляется в сторону нуля, абсолютное значе- ние остатка всегда меньше абсолютного значения делителя.

При 8-битном источнике (байт), используется 16-битовое делимое (AX). В результате деления получается два числа: частное помещается в регистр AL, а остаток – в AH.

При 16-битовом делителе (слово), используется 32-битовое делимое (DX:AX, причем DX содержит старшую значимую часть, а регистр AX – младшую). Команда деления помещает частное в регистр AX, а остаток в DX.

Значения флагов CF, OF, SF, ZF, AF и PF после этой команды не опреде- лены, а переполнение (если частное больше того, что может быть помещено в регистр результата (255 для байтового деления и 65535 для деления слов)) или деление на ноль вызывает прерывание 0h.

IDIV источник

Команда IDIV выполняет целочисленное деление со знаком AL, АХ или ЕАХ (в зависимости от размера источника) на источник (регистр или перемен- ная) и помещает результат в AL, АХ или ЕАХ, а остаток — в АН, DX или EDX со- ответственно. В остальном IDIV соответствует DIV.

* 1. *Процедуры.*

Процедура в ассемблере – это аналог функции C, процедур и функций PASCAL 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.

Описание операндов PROC:

-- *метка* – название процедуры.

-- *тип* может принимать значения NEAR и FAR, и если он указан, все коман- ды 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, опре- деляя взаимодействие процедуры с языками высокого уровня. В некоторых ас- семблерах 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 пара- метров.

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

Параметры в процедуры можно передавать в регистрах, в глобальных пе- ременных, в стеке, в потоке кода, в блоке параметров. Одна из простых передач

– передача параметров через регистры:

При передаче параметров в стеке, для чтения параметров из стека в про- цедуре 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 сме- щение параметра от начала активационной записи (то есть от ВР), например, так:

**Код программы**

.8086

.model small

.stack 100h

.data

message1 db "Enter size of array:$"

message2 db "Enter element of array:$"

sign db "+"

overflow\_message db "Overflow detected!$"

not\_digit\_message db "There are not digits in your input$"

correction\_message db "Enter correct number:$"

result\_message db "The most popular number in array is $"

input\_string\_buffer db 200 dup ("$")

number\_buffer db 9 dup("$")

crlf db 0Dh, 0Ah, '$'

arr dw 200 dup(?)

size dw ?

.code

output\_string PROC

push bp

mov bp, sp

mov ah, 09h

mov dx, [bp + 4]

int 21h

pop bp

ret 2

output\_string ENDP

input\_number PROC

push bp

mov bp, sp ;[bp + 4] input\_string\_buffer

input:

mov ah, 0Ah

mov dx, [bp + 4]

int 21h

xor ax, ax ;res

mov si, [bp + 4] ;input\_string\_buffer

mov cl, [si + 1] ;length

add si, 2

mov bl, [si]

cmp bl, '-'

je set\_negative\_sign

parse\_loop:

xor bx, bx

mov bl, [si]

sub bl, 48

cmp bl, 0

jb handle\_not\_digit

cmp bl, 9

ja handle\_not\_digit

push bx

mov bx, 10

imul bx

pop bx

jo handle\_overflow ;amount of nums is big

xor bh, bh

add ax, bx

jo handle\_overflow ;after big number

inc si

loop parse\_loop

lea di, sign

mov dl, '-'

cmp [di], dl

je set\_negative

jmp exit

handle\_not\_digit:

mov ah, 09h

lea dx, crlf

int 21h

mov ah, 09h

mov dx, offset not\_digit\_message

int 21h

jmp reload

handle\_overflow:

mov ah, 09h

lea dx, crlf

int 21h

mov ah, 09h

lea dx, overflow\_message

int 21h

jmp reload

set\_negative\_sign:

lea di, sign

mov bl, '-'

mov [di], bl

inc si

dec cl

jmp parse\_loop

reload:

mov bl, '+'

lea di, sign

mov [di], bl

xor ax, ax

xor bx, bx

xor cx, cx

xor dx, dx

xor di, di

xor si, si

mov ah, 09h

lea dx, crlf

int 21h

mov ah, 09h

lea dx, correction\_message

int 21h

jmp input

set\_negative:

neg ax

exit:

mov bl, '+'

lea di, sign

mov [di], bl

pop bp

ret 2

input\_number ENDP

output\_number PROC

push bp

mov bp, sp ;[bp + 4] bx

;[bp + 6] number\_buffer

mov ax, [bp + 4]

mov di, [bp + 6]

test ax, ax

jns positive

mov dl, '-'

mov [di], dl

inc di

neg ax

positive:

xor bx, bx

mov bx, 10

xor cx, cx

convert\_loop:

xor dx, dx

div bx

add dl, 30h

push dx

inc cx

cmp ax, 0

je final

jmp convert\_loop

final:

xor bx, bx

fill\_buffer:

pop si

add di, bx

mov [di], si

sub di, bx

inc bx

loop fill\_buffer

mov ah, 09h

mov dx, [bp + 6]

int 21h

pop bp

ret 4

output\_number ENDP

fill\_array PROC

push bp

mov bp, sp ;[bp + 4] array

;[bp + 6] input\_string\_buffer

;[bp + 8] size

xor cx, cx

xor si, si

xor bx, bx

fill\_loop:

lea dx, message2

push dx

call output\_string

push bx

push cx

push [bp + 6]

call input\_number

mov dx, ax

pop cx

pop bx

push dx

lea dx, crlf

push dx

call output\_string

pop dx

mov di, [bp + 4]

mov [bx + di], dx

add bx, 2

inc cx

mov si, [bp + 8]

cmp cx, [si]

jb fill\_loop

pop bp

ret 6

fill\_array ENDP

find\_popular PROC

push bp

mov bp, sp ;[bp + 4] arr

;[bp + 6] size

sub sp, 2 ;[bp - 2] temp

xor bx, bx

mov si, [bp + 6]

mov ax, [si]

mov bl, 2

imul bx

sub ax, 2

mov [si], ax ;size \* 2

mov di, [bp + 4];arr

mov ax, [di] ;max

xor cx, cx ;maxCounter

mov cl, 1

xor bx, bx

mov bl, 2 ;i

first\_elem:

cmp [di+bx], ax

jne skip\_inc ;arr[i] != max

inc cx ;maxCounter++

skip\_inc:

cmp bx, [si] ;i < length

je other\_elements

add bx, 2 ;i++

jmp first\_elem

other\_elements:

xor bx, bx

mov bl, 2 ;i

check:

cmp [di + bx], ax

je skip

xor dx, dx ; j

mov [bp - 2], 0

cycl:

push si

mov si, di

add si, dx

push dx

mov dx, [si]

cmp [bx + di], dx

pop dx

pop si

je operations

cmp dx, [si]

je last\_check

add dx, 2

jmp cycl

operations:

add [bp - 2], 1 ;temp++

cmp dx, [si]

je last\_check

add dx, 2

jmp cycl

last\_check:

cmp [bp - 2], cx

ja swap

skip:

cmp bx, [si]

je done

add bx, 2

jmp check

swap:

mov cx, [bp - 2]

mov ax, [bx + di]

jmp skip

done:

mov sp, bp

pop bp

ret 4

find\_popular ENDP

start:

mov ax, @data

mov ds, ax

lea dx, message1

push dx

call output\_string

lea dx, input\_string\_buffer

push dx

call input\_number

mov [size], ax

lea dx, crlf

push dx

call output\_string

push offset size

push offset input\_string\_buffer

push offset arr

call fill\_array

push offset size

push offset arr

call find\_popular

push ax

lea dx, result\_message

push dx

call output\_string

pop ax

lea dx, number\_buffer

push dx

push ax

call output\_number

mov ax, 4C00h

int 21h

end start

**Вывод программы:**

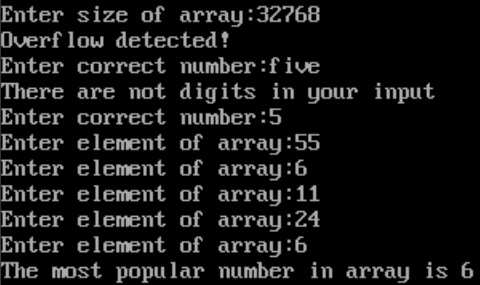
****

Рисунок 1 – Результат работы программы (.exe)