МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ
ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ГРАЖДАНСКОЙ АВИАЦИИ

Н.И. Черкасова

ПРОГРАММИРОВАНИЕ НА МАШИННО-ОРИЕНТИРОВАННЫХ ЯЗЫКАХ

Учебно-методическое пособие
по выполнению лабораторной работы № 3

для студентов II курса
направления 09.03.01
очной формы обучения

Москва · 2022

ФЕДЕРАЛЬНОЕ АГЕНТСТВО ВОЗДУШНОГО ТРАНСПОРТА
(РОСАВИАЦИЯ)

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ
УНИВЕРСИТЕТ ГРАЖДАНСКОЙ АВИАЦИИ» (МГТУ ГА)

Кафедра вычислительных машин, комплексов, систем и сетей

Н.И. Черкасова

ПРОГРАММИРОВАНИЕ НА МАШИННО-ОРИЕНТИРОВАННЫХ ЯЗЫКАХ

Учебно-методическое пособие
по выполнению лабораторной работы № 3

для студентов II курса
направления 09.03.01
очной формы обучения

Москва
ИД Академии Жуковского
2022

УДК 004.431
ББК 6Ф7.3
Ч-48

Рецензент:
Феоктистова О.Г. – д-р техн. наук, доцент

Черкасова Н.И.
Ч-48 Программирование на машинно-ориентированных языках [Текст] : учебно-методическое пособие по выполнению лабораторной работы № 3 / Н.И. Черкасова. – М.: ИД Академии Жуковского, 2022. – 32 с.

Данное учебно-методическое пособие издается в соответствии с рабочей программой учебной дисциплины «Программирование на машинно-ориентированных языках» по учебному плану направления 09.03.01 для студентов II курса очной формы обучения.

Рассмотрено и одобрено на заседаниях кафедры 28.06.2022 г. и методического совета 28.06.2022 г.

УДК 004.431
ББК 6Ф7.3

В авторской редакции

Подписано в печать 24.10.2022 г.
Формат 60x84/16 Печ. л. 2 Усл. печ. л. 1,86
Заказ № 922/0829-УМП02 Тираж 30 экз.

Московский государственный технический университет ГА
125993, Москва, Кронштадтский бульвар, д. 20

Издательский дом Академии имени Н. Е. Жуковского
125167, Москва, 8-го Марта 4-я ул., д. 6А
Тел.: (495) 973-45-68
E-mail: zakaz@itsbook.ru

ЛАБОРАТОРНАЯ РАБОТА№3
Макросредства Ассемблера

1. Цель лабораторной работы

Целью лабораторной работы является освоение:

1. Структуры и состав подпрограмм – процедур
2. Приемов работы с использованием подпрограмм – процедур. Вызов и возврат.
3. Приемов работы с использованием стека.
4. Составление приложений, обслуживающих макросредства Ассемблера.

2.Содержание отчёта

Отчет по лабораторной работе должен включать:
1) цель лабораторной работы;
2) конкретный вариант задания на выполнение;
3) тексты программ;
4) схемы алгоритмов;
5) результаты выполнения программ.

3. Краткие теоретические сведения.

3.1. Макросредства Ассемблера.

При программировании бывает полезным предварительное (до начала трансляции) преобразование текста программы. Например, может потребоваться, чтобы какой-то фрагмент программы был продублирован несколько раз или чтобы в зависимости от некоторых условий в тексте программы были сохранены одни фрагменты и удалены другие. Подобную возможность предоставляют так называемые макросредства. Расширение языка Ассемблера за счет этих средств называют макроязыком[1].

Отметим, что в общем случае, в языке ассемблера есть несколько средств, решающих проблему дублирования участков программного кода. К ним относятся:

1. макроассемблер;
2. механизм процедур;
3. механизм прерываний.

В языках ассемблера, а также в некоторых других языках программирования, макрос - символьное имя, заменяемое при обработке препроцессором на последовательность программных инструкций

Возможность дублирования предоставляют **макросредства.** Расширение языка Ассемблера за счет этих средств называют **макроязыком**[2]**.** Следует отметить, что программа, написанная на макроязыке, транслируется в два этапа.

1. Первоначально программа переводится на «чистый» ассемблер, то есть преобразуется к такому виду, где нет макросредств. Этот этап называется этапом **макрогенерации**, его осуществляет специальный транслятор - макрогенератор.
2. На втором этапе полученная программа переводится в машинный код. Этот этап называется **ассемблирование,** его осуществляет ассемблер.

Трансляция программы, написанная на макроязыке, занимает больше времени, чем трансляция программы на чистом языке ассемблера. Но макроязык предоставляет больше возможностей, поэтому писать большие программы проще на макроязыке. Макрогенератор и ассемблер могут взаимодействовать разными способами, в частности:

1.Действия независимо друг от друга: сначала текст программы обрабатывает макрогенератор и только затем начинает работать ассемблер. Такой способ достаточно прост для понимания и для реализации используется достаточно часто, однако, у него есть серьезный недостаток: макрогенератор, работая до ассемблера, не может воспользоваться информацией, извлекаемой из текста программы ассемблером. Например, в макросредствах нельзя использовать то, что после директивы N equ 10, где имя **N** обозначает число 10, так как эта директива относится к чистому ассемблеру и будет обработана

только им. Из-за подобного рода ограничений приходится усложнять макроязык.

2.Макрогенератор и ассемблер могут действовать совместно, чередуя свою работу: первым каждое предложение программы просматривает макрогенератор; если эта конструкция макроязыка, то макрогенератор соответствующим образом преобразует ее в группу предложений ассемблера и передает их на обработку ассемблеру, а если это предложение ассемблера, то макрогенератор сразу передает его ассемблеру. В таких условиях в конструкциях макроязыка уже можно ссылаться на объекты (например, константы) «чистого ассемблера».

Стандартные функции могут быть определены в библиотеке макрокоманд и доступны при дальнейшем программировании. Число операций повторного кодирования уменьшается, уменьшается и количество ошибок.

Рассмотрим некоторые особенности использования макрогенераторов. С помощью блока повторения можно один раз описать некоторый фрагмент программы, который затем многократно копируется макрогенератором. Но блоки повторения можно использовать, только если эти копии расположены рядом друг с другом.

Если фрагмент программы должен повторяться в разных местах программы используются макросы - специальным образом описанный фрагмент программы, которому дали имя, а затем в нужных местах программы выписывается ссылка на этот макрос с указанием его имени. Когда макрогенератор просматривает текст программы и встречает такую ссылку, то он вместо нее подставляет в окончательный текст программы сам макрос - соответствующий фрагмент программы. Так делается для каждой ссылки на макрос, в каком бы месте программы она не встретилась.

Наконец, введем терминологию:

1. *Макроопределение* - описание макроса,
2. ссылка на макрос - *макрокоманда*,
3. процесс замены макрокоманды на макрос - *макроподстановка*,
4. результат макроподстановки -*макрорасширение*.

Макроопределения

Описание макроса имеет следующий вид:

<имя макроса> MACRO <формальные параметры через запятую>
<тело макроса>
ENDM

Первая строка макроопределения - это директива **MACRO**, которую принято называть заголовком макроса.

В ней указывается имя, которое дается макросу, а через запятую перечисляются формальные параметры макроса. Необходимость в параметрах вызвана тем, что в общем случае макрос должен копироваться не в

неизменном виде, а с некоторыми модификациями; параметры и обозначают те величины, которые влияют на модификации.

Формальным параметрам можно давать любые имена, эти имена локализуются в теле макроса; если имя параметра совпало с именем другого объекта программы, то внутри макроопределения под этим именем понимается параметр, а не объект.

Тело макроса - это тот фрагмент программы, который затем многократно копируется. Тело может состоять из любого числа любых предложений, в которых можно использовать формальные параметры макроса. Как и в блоках повторения, формальные параметры могут обозначать любые части предложений тела. Если рядом с параметром надо указать имя или число, или если параметр надо указать внутри строки, то следует использовать макрооператор &. В теле макроса можно использовать комментарии, начинающиеся с двух точек с запятой.

Завершает макроопределение директива **ENDM** (*end of macro*). В этой директиве, в отличие от блока повторения, не надо повторять имя макроса. Если перед директивой **ENDM** указать имя макроса, то это предложение будет рассматриваться как *рекурсивный вызов макроса*.

Макроопределения могут быть размещены в любом месте текста программы, по ним в машинную программу ничего не записывается. Но макроопределение должно быть записано до первой ссылки на этот макрос - действует правило ассемблера: «Сначала опиши макрос и только затем обращайся к нему».

Макрокоманды

В тех местах программы, где макрогенератор должен подставить макрос, выписывается обращение к макросу в виде макрокоманды, которая записывается следующим образом:

<имя макроса> <фактические параметры через запятую или пробел>

3.2. Примеры использования макросов

Одним из недостатков программирования на языке ассемблера является его чрезвычайная низкоуровневость операций. Например, язык ассемблера может сложить два числа, а вот для сложения трех чисел необходимо писать специальную программу. Такое сведение к операциям низкого уровня приходится делать для любого алгоритма каким бы сложным он не был. В какой-то степени этот недостаток устраняется с помощью макросов. Следует в виде макросов описать более крупные операции, а затем составить программу с использованием макросов. Рассмотрим некоторые примеры использования макросов.

Пример 1.

В программе многократно встречается условный переход

«IF X < Y THEN GOTO L»

Эта операция реализуется тремя командами:

IF_LESS MACRO X,Y,L
MOV AX,X
CMP AX,Y
JL L
ENDM

Чтобы каждый раз не выписывать представленную группу команд, необходимо описать ее как макрос, а затем пользоваться этим макросом. Имея макрос **IF_LESS**, можно следующим образом описать вычисление минимума трех чисел:

До макрогенерации	**После макрогенерации**

MOV DX,A
IF_LESS A,B,M1
MOV DX,B
M1: IF_LESS DX,C,M2
MOV DX,C
M2:
. . .
	MOV DX,A
	MOV AX,A
	CMP AX,B
	JL M1
	MOV DX,B
	M1: MOV AX,DX
	CMP AX,C
	L M2
	MOV DX,C
	M2: . . .

Использование макросов сокращает размеры исходного текста программы и позволяет составлять программу в терминах более крупных операций. Если в виде макросов описать все часто используемые операции, то можно построить новый язык программирования, создавать программы на котором существенно легче, чем на чистом языке ассемблера. Следует отдать должное возможности программировать на самой грани между программными и аппаратными средствами.

Пример 2.

Имеется процедура вычисляющая наибольший общий делитель (**NOD**). Параметр *X* передается через регистр **AX**, параметр *Y*- через регистр **BX**, результат *Z* возвращается через **AX**. Вычисляем **CX=NOD(A,B)+NOD(C,D)**.

Соответствующий фрагмент программы следующий:

MOV AX,A
MOV BX,B
CALL NOD
MOV CX,AX
MOV CX,AX

```
MOV BX,D
CALL NOD
ADD CX,AX
```

При каждом обращении к процедуре NOD выписывается почти одна и та же группа команд.

Опишем эту группу как макрос, а затем его используем. Описание этого макроса:

```
CALL_NOD MACRO X,Y
MOV AX,X
MOV BX,Y
CALL NOD
ENDM
```

Нужный нам фрагмент программы выглядит более наглядно и короче:

```
CALL_NOD
MOV CX,AX
CALL_NOD
ADD CX,AX
```

После макроподстановок получится тот же текст программы, что и прежде, но построением такого текста будет заниматься макрогенератор, а не программист.

Для примера использования макроопределений также рассмотрим программу вывода имени в диалоговое окно.

```
.686P
MODEL FLAT,
STDCALL
PRINTNAME MACRO NAME
LOCAL STR1, STR2, METKA
JMP METKA
STR1 DB «ПРОГРАММА»,0
STR2 DB «МЕНЯ ЗОВУТ: &NAME «,0
METKA:
PUSH 0
PUSH OFFSET STR1
PUSH OFFSET STR2
PUSH 0
CALL MESSAGEBOXA@16
ENDM
INIT MACRO
EXTERN MESSAGEBOXA@16:NEAR
ENDM
INIT
```

```
.CODE
START:
PRINTNAME <ЛЕНА>
PRINTNAME <ТАНЯ>
RET
END START
```

Результат выполнения

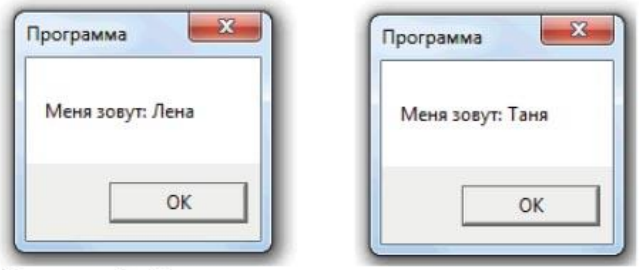

Рисунок 1- Результат выполнения программы

3.2.1. Варианты размещения макроопределений

Макрокоманды в ассемблере схожи с директивой **#define** в языке Си.
Существует три варианта размещения макроопределений:

1. в начале исходного текста программы, до кода и данных с тем, чтобы не ухудшать читаемость программы. Этот вариант следует применять в случаях, если определяемые макрокоманды актуальны только в пределах одной этой программы;
2. в отдельном файле. Этот вариант подходит при работе над несколькими программами одной проблемной области. Чтобы сделать доступными эти макроопределения в конкретной программе, необходимо в начале исходного текста этой программы записать директиву INCLUDE ИмяФайла, например:

```
.586
.MODEL FLAT,
STDCALL
INCLUDE SHOW.INC ;сюда вставляется текст файла show.inc
```

3. в макробиблиотеке. Универсальные макрокоманды, которые используются практически во всех программах целесообразно записать в так называемую макробиблиотеку. Сделать актуальными макрокоманды из этой библиотеки можно также с помощью директивы include. Недостаток этого и предыдущего способов в том, что в исходный текст программы включаются абсолютно все макроопределения. Для исправления ситуации можно использовать директиву purge, в качестве операндов которой через запятую

перечисляются имена макрокоманд, которые не должны включаться в текст программы. К примеру:

```
INCLUDE IOMAC.INC
PURGE OUTSTR,EXIT
```

В данном случае в исходный текст программы перед началом трансляции MASM вместо строки INCLUDE IOMAC.INC вставит строки из файла IOMAC.INC. Но вставленный текст будет отличаться от оригинала тем, что в нем будут отсутствовать макроопределения OUTSTR и EXIT.

Пример.

Библиотека макроопределений представляет собой файл. Его содержимое можно считывать в любую исходную программу. Тем самым все макроопределения библиотеки становятся доступными для этой программы. Чтобы использовать какое-либо из них, достаточно точно указать его имя.

Структура программы с макроопределениями (листинг 1).

Листинг 1

```
ADD_WORDS MACRO TERM1, TERM2, SUM
MOV AX, TERM1
ADD AX, TERM2
MOV SUM, AX
ENDM

SSS SEGMENT STACK
DB 128 DUP (?)
SSS ENDS

DSS SEGMENT
X1 DB 13
X2 DB 11
YS DB ?
A1 DB 44
B1 DB 33
AB DB ?
DSS ENDS

CSEG SEGMENT
ASSUME CS:CSEG,DS:DSS,SS:SSS
PUSH AX
MOV BX,DSS
MOV DS,BX
BEG PROC
ADD_WORDS X1,X2,YS
ADD_WORDS A1,B1,AB
RETF
BEG ENDP
```

```
CSEG ENDS
END BEG
```

Итак, для считывания библиотеки макроопределений в исходную программу надо передать Ассемблеру ее имя в операторе INCLUDE, но чтобы избежать повторения считывания, оператор INCLUDE надо поместить в условную структуру IF1, например:

```
IF1
INCLUDE MACRO.LIB
ENDIF
```

Это заставит Ассемблер считать библиотеку во время прохода 1. Но при этом, указанный в операторе INCLUDE текст в листинг не попадет, поскольку Ассемблер выдает его во время прохода 2.

```
INCLUDE MACRO.LIB
PURGE MAC1, MAC2, MAC3
```

Функционально макроопределения похожи на процедуры. Сходство их в том, что и те, и другие достаточно один раз описать, а затем вызывать их многократно специальным образом. Различия макроопределений и процедур в зависимости от целевой установки можно рассматривать и как достоинства, и как недостатки:

- в отличие от процедуры, текст которой неизменен, макроопределение в процессе макрогенерации может меняться в соответствии с набором фактических параметров. При этом коррекции могут подвергаться как операнды команд, так и сами команды. Процедуры в этом отношении менее гибки;
- при каждом вызове макрокоманды ее текст в виде макрорасширения вставляется в программу. При вызове процедуры микропроцессор осуществляет передачу управления на начало процедуры, находящейся в некоторой области памяти в одном экземпляре. Код в этом случае получается более компактным, хотя быстродействие несколько снижается за счет необходимости осуществления переходов

3.3. Процедуры

3.3.1. Определение процедуры

Процедура (подпрограмма) — это основная функциональная единица декомпозиции (разделения на несколько частей) некоторой задачи [2,3]. Процедура представляет собой группу команд для решения конкретной подзадачи и обладает средствами получения управления из точки вызова задачи более высокого приоритета и возврата управления в эту точку. В простейшем случае программа может состоять из одной процедуры.

Процедуру можно определить и как правильным образом оформленную совокупность команд, которая, будучи однократно описана, при необходимости может быть вызвана в любом месте программы. Функция – процедура, способная возвращать некоторое значение.

Процедуры ценны тем, что могут быть активизированы в любом месте программы. Процедурам могут быть переданы некоторые аргументы, что позволяет, имея одну копию кода в памяти и изменять ее для каждого конкретного случая использования, подставляя требуемые значения аргументов.

Для описания последовательности команд в виде процедуры в языке ассемблера используются две директивы: PROC и ENDP.

Синтаксис описания процедуры:

ИмяПроцедуры PROC расстояние;
тело процедуры
ИмяПроцедуры ENDP

В заголовке процедуры (директиве PROC) обязательным является только задание имени процедуры. Атрибут расстояние может принимать значения near или far и характеризует возможность обращения к процедуре из другого сегмента кода. По умолчанию атрибут расстояние принимает значение near, и именно это значение используется при выборе плоской модели памяти FLAT.

3.3.2. Расположение процедуры

Процедура может размещаться в любом месте программы, но так, чтобы на нее случайным образом не попало управление. Если процедуру просто вставить в общий поток команд, то микропроцессор будет воспринимать команды процедуры как часть этого потока. Учитывая это обстоятельство, есть следующие варианты размещения процедуры в программе:

1. в начале программы (до первой исполняемой команды);
2. в конце (после команды, возвращающей управление операционной системе);
3. промежуточный вариант — тело процедуры располагается внутри другой процедуры или основной программы;
4. в другом модуле.

Размещение процедуры в начале сегмента кода предполагает, что последовательность команд, ограниченная парой директив PROC и ENDP, будет размещена до метки, обозначающей первую команду, с которой начинается выполнение программы. Эта метка должна быть указана как параметр директивы END, обозначающей конец программы:

Листинг 2.

```
.CODE
MYPROC PROC NEAR
RET
MYPROC ENDP
START PROC
CALL MYPROC
…
START ENDP
END START
```

В этом фрагменте после загрузки программы в память управление будет передано первой команде процедуры с именем start.

Объявление имени процедуры в программе равнозначно объявлению метки, поэтому директиву PROC в частном случае можно рассматривать как форму определения метки в программе.

Размещение процедуры в конце программы предполагает, что последовательность команд, ограниченная директивами PROC и ENDP, будет размещена после команды, возвращающей управление операционной системе.

Листинг 3.

```
.CODE
START PROC
CALL MYPROC
…
START ENDP
MYPROC PROC NEAR
RET
MYPROC ENDP
END START
```

Промежуточный вариант расположения тела процедуры предполагает ее размещение внутри другой процедуры или основной программы. В этом случае требуется предусмотреть обход тела процедуры, ограниченного директивами PROC и ENDP, с помощью команды безусловного перехода jmp:

Листинг 4.

```
.CODE
START PROC
JMP ML
MYPROC PROC NEAR
RET
MYPROC ENDP
ML:
…
```

START ENDP
END START

Последний вариант расположения описаний процедур - в отдельном модуле — предполагает, что часто используемые процедуры выносятся в отдельный файл. Файл с процедурами должен быть оформлен как обычный исходный файл и подвергнут трансляции для получения объектного кода. Впоследствии этот объектный файл на этапе компоновки объединяется с файлом, в котором эти процедуры используются. Этот способ предполагает наличие в исходном тексте программы еще некоторых элементов, связанных с особенностями реализации концепции модульного программирования в языке ассемблера. Вариант расположения процедур в отдельном модуле используется также при построении Windows-приложений на основе вызова API-функций.

Поскольку имя процедуры обладает теми же атрибутами, что и метка в команде перехода, то обратиться к процедуре можно с помощью любой команды условного или безусловного перехода. Но благодаря специальному механизму вызова процедур можно сохранить информацию о контексте программы в точке вызова процедуры. Под контекстом понимается информация о состоянии программы в точке вызова процедуры. В системе команд микропроцессора есть две команды, осуществляющие работу с контекстом. Это команды CALL И RET:

CALL ИмяПроцедуры@num — вызов процедуры (подпрограммы).
RET число — возврат управления вызывающей программе.
число — необязательный параметр, обозначающий количество байт, удаляемых из стека при возврате из процедуры.

@num – количество байт, которое занимают в стеке переданные аргументы для процедуры (параметр является особенностью использования транслятора MASM).

3.3.3. Объединение процедур, расположенных в разных модулях

Особого внимания заслуживает вопрос размещения процедуры в другом модуле. Так как отдельный модуль - это функционально автономный объект, то он ничего не знает о внутреннем устройстве других модулей, и наоборот, другим модулям также ничего не известно о внутреннем устройстве данного модуля. Но каждый модуль должен иметь такие средства, с помощью которых он извещал бы транслятор о том, что некоторый объект (процедура, переменная) должен быть видимым вне этого модуля. И наоборот, нужно объяснить транслятору, что некоторый объект находится вне данного модуля. Это позволит транслятору правильно сформировать машинные команды, оставив некоторые их поля незаполненными. Позднее, на этапе компоновки

настраивает модули и разрешает все внешние ссылки в объединяемых модулях.

Для того чтобы объявить о видимых извне объектах, программа должна использовать две директивы MASM: EXTERN и PUBLIC. Директива EXTERN предназначена для объявления некоторого имени внешним по отношению к данному модулю. Это имя в другом модуле должно быть объявлено в директиве PUBLIC. Директива PUBLIC предназначена для объявления некоторого имени, определенного в этом модуле и видимого в других модулях.

Синтаксис этих директив следующий:

EXTERN имя:тип, …, имя:тип
PUBLIC имя, …, имя

Здесь имя - идентификатор, определенный в другом модуле. В качестве идентификатора могут выступать:

1. имена переменных;
2. имена процедур;
3. имена констант.

Тип определяет тип идентификатора. Указание типа необходимо для того, чтобы транслятор правильно сформировал соответствующую машинную команду. Действительные адреса будут вычислены на этапе компоновки, когда будут разрешаться внешние ссылки. Возможные значения типа определяются допустимыми типами объектов для этих директив:

1. если имя - это имя переменной, то тип может принимать значения BYTE, WORD, DWORD, QWORD И TBYTE;
2. если имя — это имя процедуры, то тип может принимать значения NEAR или FAR;
3. в компиляторе MASM после имени процедуры необходимо указывать число байтов в стеке, которые занимают аргументы функции: EXTERN P1@0:NEAR
4. если имя — это имя константы, то тип должен быть ABS.

Пример использования директив EXTERN и PUBLIC для двух модулей

```
;Модуль 1
.586
.MODEL FLAT, STDCALL
.DATA
EXTERN P1@0:NEAR
.CODE
START PROC
CALL P1@0
RET
START ENDP
```

```
END START
;МОДУЛЬ 2
.586
.MODEL FLAT, STDCALL
PUBLIC P1
.DATA
.CODE
P1 PROC
RET
P1 ENDP
END
```

Исполняемый модуль находится в программе Модуль 1, поскольку содержит метку START, с которой начинается выполнение программы (эта метка указана после директивы End в программе Модуль 1). Программа вызывает процедуру P1, внешнюю, содержащуюся в файле Модуль 2. Процедура P1 не имеет аргументов, поэтому описывается в программе Модуль1 с помощью директивы

```
EXTERN P1@0:NEAR
```

@0 – количество байт, переданных функции в качестве аргументов
NEAR – тип функции (для плоской модели памяти всегда имеет тип near).

Вызов процедуры осуществляется командой;

```
CALL P1@0
```

При входе в процедуру приходится сохранять в стеке содержимое регистров, для чего многократно записывается несколько команд **PUSH**, а при выходе из процедуры несколько команд **POP**. Если в программе много процедур следует записать эти команды в виде макросов.

У этих макросов может быть любое количество фактических параметров (разное количество названий регистров), так как в языке ассемблера можно определять макросы с фиксированным числом формальных параметров, поэтому макрос описывается с одним формальным параметром, но при обращении к нему в макрокоманде указывают через запятую нужное количество фактических параметров и заключают весь список в угловые скобки, в результате получается синтаксически один параметр. В теле макроса от этого списка отделяют по одному настоящему параметру и что-то с ним делают.

3.4. Передача параметров процедуры

Существуют несколько способов передачи параметров в процедуру.

3.4.1. Передача через регистры

Если процедура получает небольшое число параметров, идеальным местом для их передачи оказываются регистры.

Существуют соглашения о вызовах, предполагающие передачу параметров через регистры ECX и EDX. Этот метод самый быстрый, но он удобен только для процедур с небольшим количествомпараметров.

Рассмотрим типичный пример.

Пусть надо вычислить

R = MAX (A, B) – MAX (B + 1, 5), где все числа знаковые и размером в слово.

Опишем процедуру, которая находит наибольшее из двух значений, находящихся в регистрах AX и BX, а результат помещает в регистр AX.

Тогда программа может быть такой:

Листинг 5.

```
; R = MAX(A, B) + MAX(B+1, 5)
.686
INCLUDE /MASM32/INCLUDE/IO.ASM
.DATA
A DW ?
B DW ?
R DW ?
.code
MAX PROC
 CMP AX, BX
 JGE RT
 MOV AX, BX
RT: RET
MAX ENDP
START:
 PRINT "A=" ; ВВОД ДАННЫХ
 ININT A
 PRINT "B=" ; ВВОД ДАННЫХ
 ININT B
 MOV AX, A
 MOV BX, B
 CALL MAX
 MOV R, AX
 Print "R=" ; Вывод Результата
 Outint16 R
 Newline
 Println "= = = = = = = ="
 MOV AX, B
 ADD AX, 1 ; [AX]=B+1
 MOV BX, 5
 CALL MAX
 ADD R, AX
```

```
 PRINT "R=" ; ВЫВОД РЕЗУЛЬТАТА
 OUTINT16 R
 INKEY ; ОЖИДАНИЕ НАЖАТИЯ КЛАВИШИ
 EXIT
END START
```

В этом примере параметры передаются по значению: перед обращением к процедуре основная процедура вычисляет значения фактических параметров и именно эти значения записывает в регистры.

Листинг 6

```
.686
INCLUDE \MASM32\INCLUDE\IO.ASM
.DATA
 X DD 10 DUP (1)
 Y DD 10 DUP (2)
 L_XY = TYPE X
 NX DD ?
 NY DD ?
.CODE
OUT_ARR1 PROC
L_PRN:
 OUTINT32 [EBX], 6
 ADD EBX, L_XY
 LOOP L_PRN
 NEWLINE
 RET
OUT_ARR1 ENDP
LSTART:
 PRINT "NX="
 ININT NX
 MOV ECX, NX
 LEA EBX, X
 CALL OUT_ARR1
 PRINT "= = = = = = = = = = ="
 NEWLINE
 PRINT "NY="
 ININT NY
 MOV ECX, NY
 LEA EBX, Y
 CALL OUT_ARR1
LEXIT:
 NEWLINE
 INKEY "PRESS ANY KEY TO EXIT."
 EXIT
END LSTART
```

3.4.2. Передача в глобальных переменных

Параметры процедуры можно записать в глобальные переменные, к которым затем будет обращаться процедура. Однако этот метод является неэффективным, и его использование может привести к тому, что рекурсия и повторная входимость станут невозможными.

Рассмотрим пример передачи параметров через глобальные переменные.

В данном случае входные данные находятся в глобальных переменных AA И BB, РЕЗУЛЬТАТ ПОМЕЩАЕТСЯ В ПОЛЕ R.

Листинг 7

```
; R = MAX(AA, BB) + MAX(BB+1, 5)
…
.DATA
…
A DW ?
B DW ?
AA DW ? ; ПАРАМЕТР
BB DW ? ; ПАРАМЕТР
R DW ?
.CODE
MAX PROC
 MOV AX, AA
 CMP AX, BB
 JGE RT
 MOV AX, BB
RT: MOV R, AX
 RET
MAX ENDP
START:
; ВВОД ДАННЫХ
…
 MOV AX, A
 MOV AA, AX ; AA=A
 MOV AX, B
 MOV BB, AX ; BB=B
 CALL MAX
 MOV BX, R
 MOV AX, B
 ADD AX, 1
 MOV AA, AX ; AA=B+1
 MOV BB, 5 ; BB=5
CALL MAX
```

```
 ADD R, BX
 ; вывод результата
…
EXIT
END START
```

ПУСТЬ ИМЕЕТСЯ ПРОЦЕДУРА НА ЯЗЫКЕ C++:

```
VOID UMN8(INT *N) {
 *N = *N * 8;
 RETURN;
}
```

Очевидно, что процедура, работающая с адресом переменной(*N), должна получать из вызывающей программы адрес своего фактического параметра для того, чтобы при необходимости изменить значение этого параметра.

Запишем процедуру UMN8, предполагая, что адрес параметра находится в регистре EBX. При этом будем считать, что фактические параметры размещаются в сегменте данных (адрес параметра может находиться в любом регистре-модификаторе, т.е.

EBX, EBP, ESI ИЛИ EDI).

Листинг 8

```
.686
INCLUDE /MASM32/INCLUDE/IO.ASM
.DATA
N DD ?
.CODE
UMN8 PROC
 SHL DWORD PTR [EBX], 3 ; X = X * 8
 RET
UMN8 ENDP
START:
 PRINT "N=" ; ВВОД ДАННЫХ
 ININT N
 LEA EBX, N ; BX=АДРЕС N
 CALL UMN8 ; UMN8(N)
 PRINT "N*8=" ; ВЫВОД РЕЗУЛЬТАТА
 OUTINT32 N
 NEWLINE
 INKEY ; ОЖИДАНИЕ НАЖАТИЯ КЛАВИШИ
 EXIT
END START
```

ПРОЦЕДУРУ UMN8 МОЖНО ПЕРЕПИСАТЬ НА АССЕМБЛЕРЕ (UMN8_ASM) ДЛЯ ОБЕСПЕЧЕНИЯ ЭФФЕКТИВНОЙ РАБОТЫ ПРОГРАММЫ.

```
VOID UMN8(INT *N) {
 *N = *N * 8;
```

```
 RETURN;
}
VOID UMN8_ASM(INT *N) {
 _ASM {
 MOV EBX, N
 SHL DWORD PTR[EBX], 3 ; N: = N * 8
 }
 RETURN;
}
INT MAIN() {
 INT N, NN; // ТЕСТИРУЕМОЕ ЧИСЛО
 COUT << "N="; CIN >> N;
 NN = N;
 UMN8(&N);
 COUT << "N=" << N << ENDL;
 UMN8_ASM(&NN);
 COUT << "NN=" << NN << endl;
…
}
```

3.4.3. Передача в блоке параметров и в стек

Блок параметров — это участок памяти, содержащий параметры и располагающийся обычно в сегменте данных.

Процедура получает адрес начала этого блока при помощи любого метода передачи параметров (в регистре, переменной, стеке, коде или даже в другом блоке параметров).

Однако, передача параметров через стек - наиболее распространённый способ. Именно его используют языки высокого уровня Параметры помещаются в стек непосредственно перед вызовом процедуры. В стек кладутся несколько параметров и затем вызывается процедура. Команда CALL также кладёт в стек адрес возврата, таким образом, адрес возврата оказывается в стеке поверх параметров. Однако поскольку в рамках своего участка стека процедура может обращаться без ограничений к любой ячейкипамяти, нет необходимости перекладывать куда-то адрес ,а потом возвращать его обратно в стек. Для обращения к первому параметру используют адрес [ESP+4] (прибавляем 4, так как на архитектуре Win32 адрес имеет размер 32 бита), для обращения ко второму параметру — адрес [ESP+8] и т.д.

В приведенном далее примере переменные A=1, B=2, R=3 помещаются в стек, процедура изменяет их значения в соответствии с адресами:

Адрес возврата PUSH R PUSH B PUSH A

… EBP=ESP ESP+4 ESP+8 ESP+12 …

После обращения к процедуре извлекаем результаты работы

в переменные RR, BB, AA.
Пример . Передача параметров через стек.
Листинг 9

```
.686
INCLUDE /MASM32/INCLUDE/IO.ASM
.DATA
 A DD 1 ; ИНИЦИАЛИЗИРУЕМ ВХОДНЫЕ ПАРАМЕТРЫ
 B DD 2
 R DD 3
 AA DD ? ; ВЫХОДНЫЕ ПАРАМЕТРЫ
 BB DD ?
 RR DD ?
 COL_POS = 5 ; КОЛИЧЕСТВО ПОЗИЦИЙ ПРИ ВЫВОДЕ
.CODE
EX_PROC_STACK PROC
 MOV EAX, [ESP+4]
 ADD EAX, 10 ; ИЗМЕНЯЕМ R
 MOV [ESP+4], EAX
 MOV EAX, [ESP+8]
 ADD EAX, 100 ; ИЗМЕНЯЕМ B
MOV [ESP+8], EAX
 MOV EAX, [ESP+12]
 ADD EAX, 1000 ; ИЗМЕНЯЕМ A
 MOV [ESP+12], EAX
 RET
EX_PROC_STACK ENDP
START:
; ВЫВОД ДАННЫХ
 PRINT " A="
 OUTINT32 A, COL_POS
 PRINT " B="
 OUTINT32 B, COL_POS
 PRINT " R="
 OUTINT32 R, COL_POS
 NEWLINE
 PUSH A ; ПОМЕЩАЕМ В СТЕК ВХОДНЫЕ ПАРАМЕТРЫ
 PUSH B
 PUSH R
 CALL EX_PROC_STACK
 POP RR ; ИЗВЛЕКАЕМ ИЗ СТЕКА ВЫХОДНЫЕ ПАРАМЕТРЫ
 POP BB
 POP AA
 ; ВЫВОД РЕЗУЛЬТАТА
 PRINT " AA="
```

```
 OUTINT32 AA, COL_POS
 PRINT " BB="
 OUTINT32 BB, COL_POS
 PRINT " RR="
 OUTINT32 RR, COL_POS
 NEWLINE
 ADD ESP, 12 ; ОСВОБОЖДАЕМ 12 БАЙТОВ СТЕКА
 INKEY ; ОЖИДАНИЕ НАЖАТИЯ КЛАВИШИ
 EXIT
END START
END
```

3.4.4.Возврат результата процедуры

Для передачи результата процедуры обычно используется регистр EAX. Этот способ используется не только в программах на языке ассемблера. Иногда удобно передать в качестве параметра адрес ячейки памяти, куда будет записан результат.

; Передача параметров через стек,
; возврат результата по адресу

Листинг 10

```
.DATA
A DD 11 ; ИНИЦИАЛИЗИРУЕМ ВХОДНЫЕ ПАРАМЕТРА
B DD 22
R DD ?
COL_POS = 5 ; КОЛИЧЕСТВО ПОЗИЦИЙ ПРИ ВЫВОДЕ
.CODE
EX_PROC_A_ADD_B PROC
 MOV EAX, [ESP+4] ; EAX = A
 MOV EBX, [ESP+8] ; EBX = B
 ADD EAX, EBX ; EAX = A + B
 MOV EDX, [ESP+12] ; EDX - АДРЕС R
 MOV [EDX], EAX ; R = EAX
 RET
EX_PROC_A_ADD_B ENDP
START:
…
 PUSH OFFSET R ; В СТЕКЕ АДРЕС ПЕРЕМЕННОЙ R
 PUSH B ; ПОМЕЩАЕМ В СТЕК ВХОДНЫЕ ПАРАМЕТРЫ
 PUSH A
 CALL EX_PROC_A_ADD_B
 PRINT " R="
 OUTINT32 R, COL_POS
 ADD ESP, 12 ; ОСВОБОЖДАЕМ 12 БАЙТ СТЕКА
```

…
END START
END

3.5. Команды работы со стеком

Стек — это область памяти, специально выделяемая для временного хранения данных программы. Важность стека определяется тем, что для него в структуре программы предусмотрен отдельный сегмент. На тот случай, если программист забыл описать сегмент стека в своей программе, компоновщик TLINK выдаст предупреждающее сообщение.

Для работы со стеком предназначены три регистра:
SS — сегментный регистр стека;
SP/ESP — регистр указателя стека;
BP/EBP — регистр указателя базы кадра стека.

Размер стека зависит от режима работы микропроцессора и ограничивается 64 Кбайт (или 4 Гбайт в защищенном режиме).
В каждый момент времени доступен только один стек, адрес сегмента которого содержится в регистре SS. Этот стек называется текущим. Для того чтобы обратиться к другому стеку («переключить стек»), необходимо загрузить в регистр SS другой адрес. Регистр SS автоматически используется процессором для выполнения всех команд, работающих со стеком.

Отметим некоторые особенности работы со стеком:

1. запись и чтение данных в стеке осуществляется в соответствии с принципом LIFO (Last In First Out – «последним пришел, первым ушел»);
2. \по мере записи данных в стек последний растет в сторону младших адресов.

Эта особенность заложена в алгоритм команд работы со стеком, при использовании регистров ESP/SP и EBP/BP для адресации памяти ассемблер автоматически считает, что содержащиеся в нем значения представляют собой смещения относительно сегментного регистра SS.

Стек используется для передачи параметров и для хранения локальных данных процедур. В принципе, для работы со стеком существуют всего две операции: **PUSH** и **POP**. Для каждой операции (положить данные и взять данные) существует несколько команд, которые отличаются тем, с какими данными они работают. Для того чтобы положить данные в стек, используется команда **PUSH**:
PUSH op
Операнд op может быть регистром, ячейкой памяти или непосредственным операндом. Размер операнда должен быть 2 или 4 байта. Операнд кладётся на вершину стека, а значение регистра ESP уменьшается на размер операнда.
Выполнение этой команды можно описать так:
ESP = [ESP] – (TYPE op); TYPE op = 2 или 4 байта

[ESP] = op

Это означает, что сначала значение регистра ESP уменьшается на 2 или 4 (вычитание происходит по модулю 216), т.е. ESP сдвигается вверх и указывает на свободную ячейку области стека, а затем в нее записывается операнд.

Флаги команда не меняет.

Для записи в стек числа его предварительно придется поместить в регистр, например:

```
MOV AX, 5
PUSH AX ; 5 -> стек
```

Для того чтобы взять данные из стека, используется команда

POP:

```
POP op
```

Операнд op может быть регистром или ячейкой памяти. Размер операнда должен быть 2 или 4 байта. В соответствии с размером операнда из вершины стека берутся 2 или 4 байта и помещаются в указанный регистр или ячейку памяти. Значение регистра ESP увеличивается на размер операнда.

Выполнение команды:

Op = [ESP]

ESP = [ESP] + (TYPE op); TYPE op = 2 или 4 байта

Кроме этих основных команд существуют ещё команды, которые позволяют сохранять в стеке и восстанавливать из стека содержимое всех регистров общего назначения, и команды, которые позволяют сохранять в стеке и восстанавливать из стека содержимое регистра флагов.

PUSHA

PUSHAD

Команда PUSHA сохраняет в стеке содержимое регистров

AX, CX, DX, BX, SP, BP, SI, DI.

Команда PUSHAD сохраняет в стеке содержимое регистров

EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI.

Для регистра (E)SP сохраняется значение, которое было до того, как положили регистры в стек. После этого значение регистра (E)SP изменяется как обычно.

POPA

POPAD

Эти команды противоположны предыдущим – они восстанавливают из стека значения регистров

(E)DI, (E)SI, (E)BP, (E)SP, (E)BX, (E)DX, (E)CX, (E)AX.

Содержимое регистра (E)SP не восстанавливается из стека, а изменяется как обычно.

PUSHF

PUSHFD

Команда PUSHF сохраняет в стеке младшие 16 бит регистра
флагов. Команда PUSHFD сохраняет в стеке все 32 бита регистра
флагов.

POPF
POPFD
Команда POPF восстанавливает из стека младшие 16 бит регистра флагов. Команда POPFD восстанавливает из стека все 32 бита регистра флагов.

3.5.1. Некоторые особенности работы со стеком

Сохранение значений регистров
Стек часто используется для временного хранения значений регистров. При организации вложенных циклов с использованием команды LOOP требуется сохранять значение регистра CX. Это значение можно сохранять в другом регистре или ячейке памяти. Однако с использованием стека это можно сделать так:

```
 PUSH CX ; CX в стек
 MOV CX, N
L: …
 LOOP L
 POP CX ; Из стека в CX
```

Пересылка данных через стек
Присваивание X = Y, где X и Y – переменные, можно реализовать так:

```
 MOV EAX, Y | PUSH Y
 MOV X, EAX | POP X
```

Проверка на выход за пределы стека
Команды PUSH и POP не осуществляют проверку на выход за пределы стека. Например, если стек пуст, и мы применяем команду чтения из стека, то ошибки не будет (считывается слово, следующее за сегментом стека). Аналогично не будет зафиксирована ошибка, если мы записываем в стек, когда он уже полон. Такие проверки, если требуется, нужно делать самостоятельно. Делаются они так:
ESP = 0? – стек пуст?
ESP = N? – стек полон?{N – размер стека в байтах)
При пустом стеке в регистре ESP находится число, равное размеру области стека в байтах.
Очистка и восстановление стека
Очистка стека от N слов осуществляется просто увеличением значения регистра ESP на 2*N:

```
 ADD ESP, 2*N ; очистка стека от N слов
```

Еще один вариант очистки стека заключается в том, что вначале следует запомнить значение указателя стека SP, до которого нужно будет делать очистку. Затем использовать его по своему усмотрению, но в конце надо просто восстановить в SP это

значение:

```
MOV EAX, ESP
… ; записи в стек
MOV ESP, EAX
```

Практически любые действия в языке ассемблера требуют использования регистров. Однако регистров очень мало и даже в небольшой программе невозможно будет разделить регистры между частями программы.

Поэтому для сохранения регистров обычно используется стек. Можно сохранить используемые регистры по одному с помощью команды PUSH или все сразу с помощью команды PUSHAD. В первом случае в конце процедуры нужно будет восстановить значения сохранённых регистров с помощью команды POP в обратном порядке. Во втором случае для восстановления значений регистров используется команда POPAD. При сохранении регистров указатель стека изменится на некоторое значение, зависящее от количества сохранённых регистров. Это нужно учитывать при вычислении адресов параметров процедуры, передаваемых через стек.

```
; Процедура получает два параметра по 4 байта
ANY_PROC PROC
 PUSHAD ; Сохраняем все регистры
 MOV EAX, [ESP+4+32] ; Извлекаем параметры
 ; из стека.
 ; Адрес вычисляется
 MOV EBX, [ESP+8+32] ; с учётом 32 байт,
 ; использованных
 ; при сохранении регистров
 …
 POPAD ; Извлекаем сохранённые регистры а потом возвращать его обратно в стек.
 RET
ANY_PROC ENDP
```

. Для обращения к первому параметру используют адрес [ESP+4] (прибавляем 4, так как на архитектуре Win32 адрес имеет размер 32 бита), для обращения ко второму параметру — адрес [ESP+8] и т.д. В приведенном далее примере переменные A=1, B=2, R=3помещаются в стек, процедура изменяет их значения в соответствии с адресами:

Адрес возврата PUSH R PUSH B PUSH A

… EBP=ESP ESP+4 ESP+8 ESP+12 …

После обращения к процедуре извлекаем результаты работы в переменные RR, BB, AA.

. Передача параметров через стек.

Листинг 11

```
.686
INCLUDE /MASM32/INCLUDE/IO.ASM
.DATA
 A DD 1 ; ИНИЦИАЛИЗИРУЕМ ВХОДНЫЕ ПАРАМЕТРЫ
 B DD 2
 R DD 3
 AA DD ? ; ВЫХОДНЫЕ ПАРАМЕТРЫ
 BB DD ?
 RR DD ?
 COL_POS = 5 ; КОЛИЧЕСТВО ПОЗИЦИЙ ПРИ ВЫВОДЕ
.CODE
EX_PROC_STACK PROC
 MOV EAX, [ESP+4]
 ADD EAX, 10 ; ИЗМЕНЯЕМ R
 MOV [ESP+4], EAX
 MOV EAX, [ESP+8]
 ADD EAX, 100 ; ИЗМЕНЯЕМ B
MOV [ESP+8], EAX
 MOV EAX, [ESP+12]
 ADD EAX, 1000 ; ИЗМЕНЯЕМ A
 MOV [ESP+12], EAX
 RET
EX_PROC_STACK ENDP
START:
; ВЫВОД ДАННЫХ
 PRINT " A="
 OUTINT32 A, COL_POS
 PRINT " B="
 OUTINT32 B, COL_POS
 PRINT " R="
 OUTINT32 R, COL_POS
 NEWLINE
 PUSH A ; ПОМЕЩАЕМ В СТЕК ВХОДНЫЕ ПАРАМЕТРЫ
 PUSH B
 PUSH R
 CALL EX_PROC_STACK
 POP RR ; ИЗВЛЕКАЕМ ИЗ СТЕКА ВЫХОДНЫЕ ПАРАМЕТРЫ
 POP BB
 POP AA
 ; ВЫВОД РЕЗУЛЬТАТА
 PRINT " AA="
 OUTINT32 AA, COL_POS
 PRINT " BB="
 OUTINT32 BB, COL_POS
```

```
 PRINT " RR="
 OUTINT32 RR, COL_POS
 NEWLINE
 ADD ESP, 12; ОСВОБОЖДАЕМ 12 БАЙТОВ СТЕКА
 INKEY; ОЖИДАНИЕ НАЖАТИЯ КЛАВИШИ
 EXIT
END START
END
```

Для передачи результата процедуры обычно используется регистр EAX. Этот способ используется не только в программах на языке ассемблера, но и в программах на языке C++.

Иногда удобно передать в качестве параметра адрес ячейки памяти, куда будет записан результат.

; Передача параметров через стек,
; возврат результата по адресу

Листинг 12

```
.DATA
A DD 11; ИНИЦИАЛИЗИРУЕМ ВХОДНЫЕ ПАРАМЕТРА
B DD 22
R DD ?
COL_POS = 5; КОЛИЧЕСТВО ПОЗИЦИЙ ПРИ ВЫВОДЕ
.CODE
EX_PROC_A_ADD_B PROC
 MOV EAX, [ESP+4]; EAX = A
 MOV EBX, [ESP+8]; EBX = B
 ADD EAX, EBX; EAX = A + B
 MOV EDX, [ESP+12]; EDX - АДРЕС R
 MOV [EDX], EAX; R = EAX
 RET
EX_PROC_A_ADD_B ENDP
START:
…
 PUSH OFFSET R; В СТЕКЕ АДРЕС ПЕРЕМЕННОЙ R
 PUSH B; ПОМЕЩАЕМ В СТЕК ВХОДНЫЕ ПАРАМЕТРЫ
 PUSH A
 CALL EX_PROC_A_ADD_B
 PRINT " R="
 OUTINT32 R, COL_POS
 ADD ESP, 12; ОСВОБОЖДАЕМ 12 БАЙТ СТЕКА
 …
END START
END
```

4.Задание на выполнение.

Варианты заданий

Написать программу с использованием макросов:

1. ввода с клавиатуры десятичного числа и:
   - перевода его в двоичное число
   - перевода его в шестнадцатиричное число.

   Максимальное число вводимых разрядов n,согласно заданию если число содержит меньшее число разрядов, то признак конца ввода -нажатие клавиши ввод. Параметр макроопределения: переменная, где находится число (слово или байт).
2. вывода двоичного (шестнадцатиричного) числа на экран в заданную позицию. Параметры макроопределения: переменная или регистр (16 битов), где находится двоичное число и позиция вывода (номер строки, номер столбца).
3. вывода двоичного представления символа на экран в заданную позицию. Параметры: номер строки, номер столбца, символ.
4. Вывести на экран на экран прямоугольник. Параметры: координаты, размеры, цвет

Литература

1.В.И.Юров .- Ассемблер, 2 издание- СПб: Питер, 2003.

2.Пирогов В. Ю.- Ассемблер для Windows. Изд. 4-е перераб. и доп. — СПб.: БХВ- Петербург, 2015.

3. И.А. Калашников. Ассемблер – это просто. Программирование на Ассемблере. «Бином», 2008 г.

Приложение 1

Опишем процедуру, которая находит наибольшее из двух значений, находящихся в регистрах AX и BX, а
результат помещает в регистр AX.
Тогда программа может быть такой:
Листинг 13

```
; R = MAX(A, B) + MAX(B+1, 5)
.686
INCLUDE /MASM32/INCLUDE/IO.ASM
.DATA
A DW ?
B DW ?
R DW ?
.CODE
MAX PROC
 CMP AX, BX
 JGE RT
 MOV AX, BX
RT: RET
MAX ENDP
START:
 PRINT "A=" ; ВВОД ДАННЫХ
 ININT A
 PRINT "B=" ; ВВОД ДАННЫХ
 ININT B
 MOV AX, A
 MOV BX, B
 CALL MAX
 MOV R, AX
 PRINT "R=" ; ВЫВОД РЕЗУЛЬТАТА
 OUTINT16 R
 NEWLINE
 PRINTLN "= = = = = = = ="
 MOV AX, B
 ADD AX, 1 ; [AX]=B+1
 MOV BX, 5
 CALL MAX
 ADD R, AX
 PRINT "R=" ; ВЫВОД РЕЗУЛЬТАТА
 OUTINT16 R
 INKEY; ОЖИДАНИЕ НАЖАТИЯ КЛАВИШИ
 EXIT
END START
```