Министерство образования Республики Беларусь

«ПОЛОЦКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМ. ЕВФРОСИНИИ ПОЛОЦКОЙ»

Факультет информационных технологий

Кафедра технологий программирования

Методические указания для выполнения лабораторной работы №4 по курсу «Конструирование программного обеспечения»

«Использование подпрограмм в языке низкого уровня»

Полоцк, 2022 г.

ЦЕЛЬ РАБОТЫ

Приобретение навыков написания подпрограмм (процедур) при программировании на языке ассемблера.

ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Процедуры. Функции

Процедура (подпрограмма) – это основная функциональная единица декомпозиции (разделения на несколько частей) некоторой задачи. 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:

<имя> proc [тип]
  <тело процедуры>
<имя> 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*.

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:

– в начале программы (до первой исполняемой команды);

– в конце (после команды, возвращающей управление операционной системе);

– промежуточный вариант – тело процедуры располагается внутри другой процедуры или основной программы;

– 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:

```
…
.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.

```
…
.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*:

```
…
.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).

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:

– имена переменных;
– имена процедур;
– 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:

– если имя – это имя переменной, то тип может принимать значения *byte*, *word*, *dword*, *qword* и *tbyte*;

– если имя – это имя процедуры, то тип может принимать значения *near* или *far*; в компиляторе MASM после имени процедуры необходимо указывать число байтов в стеке, которые занимают аргументы функции: *extern* p1@0:*near*

– 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 *Модуль 1* 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
```

Организация интерфейса с процедурой

Аргумент – это ссылка на некоторые данные, которые требуются для выполнения возложенных на модуль функций и размещенных вне этого модуля.

По аналогии с макрокомандами рассматривают понятия ***формального*** и ***фактического*** аргументов. Исходя из этого, формальный аргумент можно рассматривать не как непосредственные данные или их адрес, а как местодержатель для действительных данных, которые будут подставлены в него с помощью фактического аргумента.

Формальный аргумент можно рассматривать как элемент интерфейса модуля, а ***фактический аргумент*** – это то, что фактически передается на место формального аргумента.

Переменные – это данные, размещенные в регистре или ячейке памяти, которые могут в дальнейшем подвергаться изменению.

Константы – данные, значения которых не могут изменяться.

Сигнатура процедуры (функции) – это имя функции, тип возвращаемого значения и список аргументов с указанием порядка их следования и типов.

Семантика процедуры (функции) – это описание того, что данная функция делает. Семантика функции включает в себя описание того, что является результатом вычисления функции, как и от чего этот результат зависит. Обычно результат выполнения зависит только от значений аргументов функции, но в некоторых модулях есть понятие состояния. Тогда результат функции может зависеть от этого состояния, и, кроме того, результатом может стать изменение состояния. Логика этих зависимостей и изменений относится к семантике функции. Полным описанием семантики функций является исполняемый код функции или математическое определение функции.

Если переменная находится за пределами модуля (процедуры) и должна быть передана в него, то для модуля она является формальным аргументом. Значение переменной передается в модуль для замещения соответствующего параметра при помощи фактического аргумента.

Как правило, один и тот же модуль можно использовать многократно для разных наборов значений формальных аргументов. Для передачи аргументов в языке ассемблера существуют следующие способы:

- через регистры;
- через общую область памяти;
- через стек;
- с помощью директив *extern* и *public*.

Передача аргументов через регистры – это наиболее простой в реализации способ передачи данных. Данные, переданные подобным способом, становятся доступными немедленно после передачи управления процедуре. Этот способ очень популярен при небольшом объеме передаваемых данных.

Ограничения на способ передачи аргументов через регистры:

- небольшое число доступных для пользователя регистров;
- нужно постоянно помнить о том, какая информация в каком регистре находится;
- ограничение размера передаваемых данных размерами регистра. Если размер данных превышает 8, 16 или 32 бита, то передачу данных посредством регистров произвести нельзя. В этом случае передавать нужно не сами данные, а указатели на них.

Передача аргументов через общую область памяти – предполагает, что вызывающая и вызываемая программы используют некоторую область памяти как общую. Для организации такой области памяти используется атрибут комбинирования сегментов *common*.

Наличие этого атрибута указывает компоновщику, как нужно комбинировать сегменты, имеющие одно имя: все сегменты, имеющие одинаковое имя в объединяемых модулях, будут располагаться компоновщиком, начиная с одного адреса оперативной памяти. Это значит, что они будут перекрываться в памяти и, следовательно, совместно использовать выделенную память. Данные в сегментах *common* могут иметь одинаковые имена. Главное – структура общих сегментов. Она должна быть идентична во всех модулях использующих обмен данными через общую память. Недостатком этого способа в реальном режиме работы микропроцессора является отсутствие средств защиты данных от разрушения, так как нельзя проконтролировать соблюдение правил доступа к этим данным.

Передача аргументов через стек наиболее часто используется для передачи аргументов при вызове процедур. Суть этого способа заключается в том, что вызывающая процедура самостоятельно заносит в стек передаваемые данные, после чего передает управление вызываемой процедуре. При передаче управления процедуре микропроцессор автоматически записывает в вершину стека 4 байта. Эти байты являются адресом возврата в вызывающую программу. Если перед передачей управления процедуре командой *call* в стек были записаны переданные процедуре данные или указатели на них, то они окажутся под адресом возврата. Стек обслуживается тремя регистрами:

- *SS* – указатель дна стека (начала сегмента стека);
- *ESP* – указатель вершины стека;
- *EBP* – указатель базы.

Микропроцессор автоматически работает с регистрами *ESS* и *ESP* в предположении, что они всегда указывают на дно и вершину стека соответственно. По этой причине их содержимое изменять не рекомендуется. Для осуществления произвольного доступа к данным в стеке архитектура микропроцессора имеет специальный регистр *EBP*. Так же, как и для регистра *ESP*, использование *EBP* автоматически предполагает работу с сегментом стека. Перед использованием этого регистра для доступа к данным стека его содержимое необходимо правильно инициализировать, что предполагает формирование в нем адреса, который бы указывал непосредственно на переданные данные. Для этого в начало процедуры рекомендуется включить дополнительный фрагмент кода. Он имеет свое название – ***пролог процедуры***. Код пролога состоит всего из двух команд:

```
push ebp
mov ebp, esp
```

Первая команда сохраняет содержимое *ebp* в стеке с тем, чтобы исключить порчу находящегося в нем значения в вызываемой процедуре. Вторая команда пролога настраивает *ebp* на вершину стека. После этого можно не волноваться о том, что содержимое esp перестанет быть актуальным, и осуществлять прямой доступ к содержимому стека.

Конец процедуры также должен содержать действия, обеспечивающие корректный возврат из процедуры. Фрагмент кода, выполняющего такие действия, имеет свое название – ***эпилог процедуры***. Код эпилога должен восстановить контекст программы в точке вызова процедуры из вызывающей программы. При этом, в частности, нужно откорректировать содержимое стека, убрав из него ставшие ненужными аргументы, передававшиеся в процедуру. Это можно сделать несколькими способами:

– используя последовательность из n команд *pop* xx. Лучше всего это делать в вызывающей программе сразу после возврата управления из процедуры;

– откорректировать регистр указателя стека esp на величину 4*n, например, командой *add esp*,NN

где NN=4*n (n — количество аргументов). Это также лучше делать после возврата управления вызывающей процедуре;

– используя машинную команду *ret* n в качестве последней исполняемой команды в процедуре, где n — количество байт, на которое нужно увеличить содержимое регистра *esp* после того, как со стека будут сняты составляющие адреса возврата. Этот способ аналогичен предыдущему, но выполняется автоматически микропроцессором.

Программа, содержащая вызов процедуры с передачей аргументов через стек:

```
.586
.model flat, stdcall
.stack 4096
.data
.code
proc_1 proc         ; начало процедуры
push ebp            ; пролог: сохранение EBP
mov ebp, esp        ; пролог: инициализация EBP
mov eax, [ebp+8]    ; доступ к аргументу 4
mov ebx, [ebp+12] ; доступ к аргументу 3
mov ecx, [ebp+16] ; доступ к аргументу 2
pop ebp             ; эпилог: восстановление EBP
ret 12
proc_1 endp
main proc
push 2
push 3
push 4
```

```
call proc_1
ret
main endp
end main
```

Для доступа к аргументу 4 достаточно сместиться от содержимого *ebp* на 8 байт (4 байта хранят адрес возврата в вызывающую процедуру, и еще 4 байта хранят значение регистра *ebp*, помещенное в стек данной процедурой), для аргумента 3 – на 12 и т. д.

Пролог и эпилог процедуры можно также заменить командами поддержки языков высокого уровня:

– Команда *enter* подготавливает стек для обращения к аргументам, имеет 2 операнда: первый определяет количество байт в стеке, используемых для хранения локальных идентификаторов процедуры; второй определяет уровень вложенности процедуры.

– Команда *leave* подготавливает стек к возврату из процедуры, не имеет операндов.

```
proc_1 proc
enter 0,0
mov eax, [ebp+8]
mov ebx, [ebp+12]
mov ecx, [ebp+16]
leave
ret 12
proc_1 endp
```

Передача аргументов с помощью директив *extern* и *public* используется в случаях, если

– оба модуля используют сегмент данных вызывающей программы;

– у каждого модуля есть свой собственный сегмент данных;

– модули используют атрибут комбинирования сегментов *public* в директиве сегментации *segment*.

Способы передачи аргументов в процедуру

В процедуру могут передаваться либо данные, либо их адреса (указатели на данные). В языке высокого уровня это называется передачей по значению и по адресу, соответственно.

Наиболее простой способ передачи аргументов в процедуру – **передача по значению**. Этот способ предполагает, что передаются сами данные, то есть их значения. Вызываемая программа получает значение аргумента через регистр или через стек. При передаче переменных через регистр или стек на их размерность

накладываются ограничения, связанные с размерностью используемых регистров или стека. При передаче аргументов по значению в вызываемой процедуре обрабатываются их копии. Поэтому значения переменных в вызывающей процедуре не изменяются.

Передача аргументов **по адресу** предполагает, что вызываемая процедура получает не сами данные, а их адреса. В процедуре нужно извлечь эти адреса тем же методом, как это делалось для данных, и загрузить их в соответствующие регистры.

После этого, используя адреса в регистрах, следует выполнить необходимые операции над самими данными. В отличие от способа передачи данных по значению, при передаче данных по адресу в вызываемой процедуре обрабатывается не копия, а оригинал передаваемых данных. Поэтому при изменении данных в вызываемой процедуре они автоматически изменяются и в вызывающей программе, так как изменения касаются одной области памяти.

Возврат результата из процедуры

В общем случае программист располагает тремя вариантами возврата значений из процедуры:

– С использованием регистров. Ограничения здесь те же, что и при передаче данных, — это небольшое количество доступных регистров и их фиксированный размер. Данный способ является наиболее быстрым, поэтому его есть смысл использовать для организации критичных по времени вызова процедур.

– С использованием общей области памяти. Этот способ удобен при возврате большого количества данных, но требует внимательности в определении областей данных и подробного документирования для устранения неоднозначностей.

– С использованием стека. Здесь, подобно передаче аргументов через стек, также нужно использовать регистр *ebp*. При этом возможны следующие варианты:
— использование для возвращаемых аргументов тех же ячеек в стеке, которые применялись для передачи аргументов в процедуру. То есть предполагается замещение ставших ненужными входных аргументов выходными данными;
— предварительное помещение в стек наряду с передаваемыми аргументами фиктивных аргументов с целью резервирования места для возвращаемого значения. При использовании этого варианта процедура, конечно же, не должна пытаться очистить стек командой *ret*. Эту операцию придется делать в вызывающей программе, например командой *pop*.

Порядок выполнения работы

Задание 1. Вывод на экран дисплея результата работы программы при помощи процедуры.

1. Для выполнения данной практической работы необходимо восстановить и отладить программу – Подсчет количества чисел, равных 0, в массиве из 10 чисел.

2. Получить объектный файл программы приложения А.

3. Отладив программу подсчета количества чисел, равных 0, сделать изменения в ней:

3.1. Директивой EXTRN MAS10:proc — объявить имя внешней процедуры;

3.2. Вставить команду загрузки результата в регистр CL;

3.3. Вставить команду CALL MAS10 — вызов подпрограммы.

4. Оттранслировать новый вариант программы, получить второй объектный файл.

5. При помощи команды TLINK скомпоновать объектные файлы программы приложения А и модифицированной программы подсчета чисел, равных 0, в исполняемую программу.

6. Запустить полученный загрузочный модуль. При правильной работе на экран будет выведен результат – сколько нулей в массиве.

Замечание: возможны самостоятельные решения данной задачи: вывода на экран результата – подсчёта количества чисел в массиве, равных 0, используя подпрограммы. Одновременно можно подсчитать и количество чисел, меньших 0Ah.

Задание 2. Выделение из программы блока обработки массива в отдельную процедуру.

1. Для выполнения данной части практической работы необходимо воспользоваться отлаженной программой подсчёта в массиве чисел, равных 0 (исходная программа первой части).

2. Модифицировать исходную программу, создав две программы – главную и подпрограмму. Подпрограмма должна содержать блок обработки массива (поиск элементов, равных 0). Передачу параметров осуществить через стек.

3. Далее осуществить действия, аналогичные пп. 4-6 предыдущего задания.

Замечание 1: можно объединить решение заданий I и II. В этом случае, имея главную программу и две процедуры, будет выведено на экран, сколько нулей имеется в исходном массиве (а также, количества чисел, меньших 0Ah).

Содержание отчета

Отчет должен включать:

а) титульный лист;

б) формулировку цели работы;

в) описание результатов выполнения задания:

- листинги программ;
- результаты выполнения программ;

г) выводы, согласованные с целью работы.

Контрольные вопросы

1. Что означает тип объединения PUBLIC?
2. Какие существуют другие типы объединений? Их назначение.
3. Назначение директивы EXTRN.
4. Назначение директивы PUBLIC.
5. Подробно разобрать в отчете, как происходит передача параметров через стек?

ПРИЛОЖЕНИЕ А

```
Model small
.code
public MAS10 ;объявление «общим » имени процедуры
MAS10 proc ;процедура вывода на экран чисел от 0 до 10
;данные передаются через регистр CL
push AX
push DX ;сохранение значения регистров в стеке
mov DH, CL ;сохранение переданных данных
mov AH, 02h ;функция DOS – «вывод символа»
mov DL, 09h ;код символа табуляции
int 21h
sub DH, 09
cmp DH, 01 ;проверка на =10
jb M2 ;если больше 10, то на «выход»
jne M1 ;если не равно 10, то «переход»
mov DL, 31h ;если равно 10, то «вывод» 1
int 21h
mov DL, 30h ;Вывод 0
int 21h
jmp M2
M1: mov DL, 30h ;преобразование в ASCII код
Add DL, CL
Int 21h ;вывод чисел от 0 до 9
M2: pop DX
Pop AX ;восстановление содержимого регистров
RET ;возврат в вызывающую программу
MAS10 endp ;конец процедуры
end
```