МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение

высшего образования

**«УЛЬЯНОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»**

Кафедра «Вычислительная техника»

Дисциплина «Организация ЭВМ и систем»

**Лабораторная работа №7**

**«Команды организации подпрограмм и программных прерываний: Вызов подпрограмм. Возврат из подпрограмм. Передача параметров»**

**Вариант 3**

Выполнили:

студенты группы ИВТАПбд-31

Кондратьев П.С.

Гаврилов С.А.

Проверила:

Лылова А. В.

Ульяновск, 2018

**Оглавление**

[1. Вызов подпрограмм 3](#_Toc530334373)

[1.1. Способы размещения подпрограмм. 3](#_Toc530334374)

[1.2. Вызов подпрограмм для архитектуры x86 6](#_Toc530334375)

[1.3. Вызов подпрограмм для архитектуры ARM 6](#_Toc530334376)

[1.4. Вызов подпрограмм для архитектуры AVR 7](#_Toc530334377)

[1.5. Объединение процедур, расположенных в разных модулях 8](#_Toc530334378)

[2. Передача параметров 10](#_Toc530334379)

[2.1. Передача аргументов через регистры 10](#_Toc530334380)

[2.2. Передача аргументов через стек 10](#_Toc530334381)

[3. Возврат из подпрограмм 13](#_Toc530334382)

[3.1. Возврат результата из процедуры 13](#_Toc530334383)

[3.1.1. С использованием регистров 13](#_Toc530334384)

[3.1.2. С использованием общей области памяти 13](#_Toc530334385)

[3.1.3. С использованием стека 13](#_Toc530334386)

[Список использованной литературы: 14](#_Toc530334387)

# **Вызов подпрограмм**

**Подпрограмма** - поименованная или иным образом идентифицированная часть компьютерной программы, содержащая описание определённого набора действий. Подпрограмма может быть многократно вызвана из разных частей программы.

В простейшем случае (в ассемблерах) подпрограмма представляет собой последовательность команд, отдельную от основной части программы и имеющие в конце специальную команду выхода из подпрограммы.

Таким образом, участок программы, к которому можно обращаться из различных мест программы для выполнения некоторых действий, называется **подпрограммой**.

Практически в любой программе, независимо от ее содержания, встречаются участки, которые требуется выполнять (возможно, с небольшими изменениями) несколько раз по ходу программы. Такие повторяющиеся участки целесообразно выделить из общей программы, оформить в виде подпрограмм и обращаться к ним каждый раз, когда в основной программе возникает необходимость их выполнения.

Подпрограмма, в зависимости от выполняемых ею функций, может требовать передачи из вызывающей программы определенных данных (называемых аргументами, или параметрами), возвращать в вызывающую программу результаты вычислений или обходиться и без того, и без другого.

Для того чтобы получить возможность возвращаться на команду, следующую за командой вызова подпрограммы, требуется запомнить адрес этой команды. Адрес возврата хранится в особых ячейках памяти. После выполнения подпрограммы необходимо осуществить переход к адресу, который записан в этих ячейках.

Реализация подпрограмм в ассемблере задача более сложная, чем их описание на языках высокого уровня. Во-первых, в ассемблере не существует формального деления подпрограмм на процедуры и функции, и то и другое называется процедурами. Содержательно, конечно, их можно разделить на функции и «чистые процедуры» в зависимости от того, вырабатывают они результат или нет. Во-вторых, для применения подпрограмм необходимо решить несколько вопросов.

# **Способы размещения подпрограмм.**

Размещать подпрограммы разрешено в любом месте основной программы, однако управление подпрограмме никогда не должно быть самопроизвольно передано. Несколько подпрограмм обычно группируют вместе. Подпрограммы целесообразно размещать либо в конце сегмента команд, либо в самом начале сегмента – перед командой, с которой должно начаться выполнение. Также подпрограммы нередко размещают в отдельном сегменте команд.

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.

Эти команды не только осуществляет передачу управления на указанный адрес, но и запоминают адрес команды, следующей за командой вызова подпрограммы.

# **Вызов подпрограмм для архитектуры x86**

Подпрограммы представляют собой метку, по которой располагается код. Заканчивается подпрограмма инструкцией ret.

Пример:

prints\_HelloWorld:

mov ah, 0x9

mov dx, hello

int 0x21

ret

Чтобы вызвать подпрограмму используется инструкция call:

call prints\_HelloWorld

# **Вызов подпрограмм для архитектуры ARM**

Рассмотрим, как происходит вызов функций на уровне архитектуры набора команд. Каждый процессор имеет регистр Program Counter, который хранит адрес инструкции, которую нужно выполнить в данный момент. При вызове подпрограммы программа должна сохранить информацию о том, куда процессор должен вернуться после завершения подпрограммы. Для этой цели используется регистр LR. Рассмотрим пример использования регистров LR и PC для вызова подпрограммы strcpy.

ADD LR, PC, #after

B strcpy

after

other code

strcpy

LDRB R2, [R1], #1

STRB R2, [R0], #1

TST R2, R2

BNE strcpy

MOV PC, LR

Здесь к регистру PC, который содержит адрес текущей инструкции, прибавляется смещение метки after. Результат этой операции записывается в регистр LR. Таким образом, мы поместили в регистр LR адрес инструкции, на которую нужно будет перейти после вызова подпрограммы.

В инструкции B (Branch) происходит переход на метку strcpy и подпрограмма начинает свою работу. Как только подпрограмма закончилась, она копирует регистр LR обратно в PC. Поскольку регистр PC используется для определения выполняемой инструкции, изменение его значения приведёт к исполнению процессором инструкции, следующей за меткой after.

Процесс вызова подпрограмм таким способом выглядит слишком громоздким, поэтому разработчики ARM ввели новую инструкцию, которая заменяет две инструкции из нашего примера. Она называется BL (Branch and Link). Эта инструкция загружает в регистр LR адрес инструкции, следующей за BL, а затем записывает в регистр PC адрес вызываемой подпрограммы.

# **Вызов подпрограмм для архитектуры AVR**

Аналогично архитектуре x86 используются команды перехода к подпрограмме CALL (ICALL, RCALL, CALL) И команда возврата из подпрограммы RET.

…

RCALL Wait ;вызов подпрограммы

RJMP Start ; Зациклим программу.

Wait: LDI R17, Delay ; Загрузили длительность задержки

M1: DEC R17 ; Уменьшили на 1

NOP ; Пустая операция

BRNE M1 ; Длительность не равна 0? Переход если не 0

RET ; возврат из подпрограммы

# **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:

;Модуль 1

.586

.model flat, stdcall

.data

extern p1@0:near

.code

start proc

call p1@0

ret

start endp

end start

# **Передача параметров**

# **Передача аргументов через регистры**

**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 стек**

**Передача аргументов через стек** наиболее часто используется для передачи аргументов при вызове процедур. Аргументы передаются справа налево, обязанность по очистке стека лежит на вызывающей стороне.

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:

* ESS — 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 ;сохраняем указатель начала стекового кадра на стеке

mov ebp, esp ;теперь началом кадра является вершина предыдущего

Первая команда сохраняет содержимое ebр 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

# **Возврат из подпрограмм**

# **Возврат результата из процедуры**

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

# **С использованием регистров**

**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**

**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**

**С использованием стека**. Здесь, подобно передаче аргументов через стек, также нужно использовать регистр ebр. 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.

# **Список использованной литературы**

1. Aluette: [Электронный ресурс] // Основы архитектуры ARM. Стек: http://alouette.su/page/arm-stack.(Дата обращения: 10.11.2018).
2. Easy Electronics: [Электронный ресурс] // AVR. Учебный курс. Подпрограммы и прерывания: http://easyelectronics.ru/avr-uchebnyj-kurs-podprogrammy-i-preryvaniya.html.(Дата обращения: 11.11.2018).
3. Программирование: [Электронный ресурс] // Процедуры и функции в ассемблере. URL: https://prog-cpp.ru/asm-proc/.(Дата обращения: 11.11.2018).
4. Ассемблер: [Электронный ресурс] // Процедуры в Ассемблере: http://av-assembler.ru/asm/afd/procedures-in-assembler.htm.(Дата обращения: 12.11.2018).