МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение высшего профессионального образования

САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ АЭРОКОСМИЧЕСКОГО ПРИБОРОСТРОЕНИЯ

АССЕМБЛЕРЫ В ИНФОРМАЦИОННЫХ СИСТЕМАХ

Методические указания к выполнению лабораторных работ

ГУАП

Санкт-Петербург

2015

Составитель: Т.В. Семененко
Рецензент д.т.н., профессор С.И. Зиатдинов

Методические указания к выполнению лабораторных работ по курсу «Ассемблеры в информационных системах» содержат краткие теоретические сведения о конструкции языка ассемблера для ПК типа IBM PC. Рассмотрены команды ассемблера и методы программирования на нем.

Указания предназначены для студентов, обучающихся направлению 09.03.02 «Информационные системы и технологии».

Подготовлены кафедрой информационно-сетевых технологий и рекомендованы к изданию редакционно-издательским советом Санкт-Петербургского государственного университета аэрокосмического приборостроения.

Редактор
Компьютерная верстка

Сдано в набор Подписано к печати Формат
Бумага офсетная. Усл.-изд.л. усл.печ.л. Тираж экз. Заказ № .

Редакционно-издательский центр ГУАП
190000, Санкт-Петербург, Б. Морская ул., 67

Введение

В настоящее время одной из основных ЭВМ для решения большинства практических задач является IBM-совместимый компьютер с процессорами Intel Pentium и совместимыми. Для написания эффективных по времени выполнения программ для данных машин необходимо знать принципы работы процессора, команды, которые он может выполнять и т.д. Процессоры Intel Pentium совместимы по командам с первым процессором этой серии – Intel 8086, поэтому программы для Intel 8086 выполняются и на процессорах Pentium. Средством, позволяющим писать программы на уровне команд процессора, является язык ассемблера.

В данном пособии рассматривается лабораторные работы, иллюстрирующие различные возможности процессора 8086, необходимые теоретические сведения для написания и отладки ассемблерных программ, а также приведена система основных команд процессора.

1. МИКРОПРОЦЕССОР INTEL 8086

1.1. Общие принципы работы МП 8086 при выполнении прикладных программ

Программы можно условно разделить на системные и прикладные. Системные программы служат для обеспечения работы операционной системы, для работы с внешними устройствами и т. д. Основной задачей прикладных программ является решение задач пользователя. В данном пособии рассматриваются элементы написания прикладных программ.

Работа ЭВМ при выполнении прикладной задачи производится в соответствии с принципами программного управления (принципами фон Неймана):

- информация кодируется в двоичном виде и разделяется на байты;
- информация хранится в оперативной памяти, причем каждый байт имеет свой номер, или адрес;
- программа представляется в виде последовательности команд, каждая из которых записывается с помощью целого числа байт;
- порядок команд однозначно определяется программой.

Каждая прикладная программа обрабатывает некоторые данные с помощью последовательности команд. Команды делятся на арифметико-логические, пересылочные, управляющие порядком следования команд и некоторые другие, используемые системными программами (см. Приложение). Арифметико-логические команды предназначены для выполнения вычислений, работы с отдельными битами и проверки различных условий. Пересылочные операции копируют данные, расположенные по заданному адресу, в другую область памяти, то есть осуществляют операции присваивания. Команды, управляющие порядком следования команд (или команды передачи управления), позволяют указать центральному процессору, какую команду необходимо выполнить следующей. С помощью команд передачи управления можно организовать цикл, условное ветвление (оператор if языков высокого уровня) или вызов подпрограммы. Без их применения программа последовательно выполняет одну команду за другой.

Большинство команд МП 8086 состоят из кода операции и одного или двух адресов данных. Данные в этом случае часто называют операндами команды. В некоторых командах адреса операндов задавать не требуется, так как адрес известен заранее или может быть вычислен внутри процессора. В таких случаях говорят, что адрес задан неявно. Всего при использовании данного МП можно обращаться к 2^{20} байтам. Диапазон возможных адресов данных называют адресным пространством процессора. Таким образом, адресное пространство Intel 8086 составляет 2^{20} = 1048576 байт, или 1 мегабайт. Байты в МП 8086 объединяются по два в слова. Слово может начинаться как с четного, так и с нечетного адреса.

Все адресное пространство разбито на блоки (сегменты) по 64 килобайта, или 65536 байт, причем сегменты могут перекрываться (рисунок 1.1).

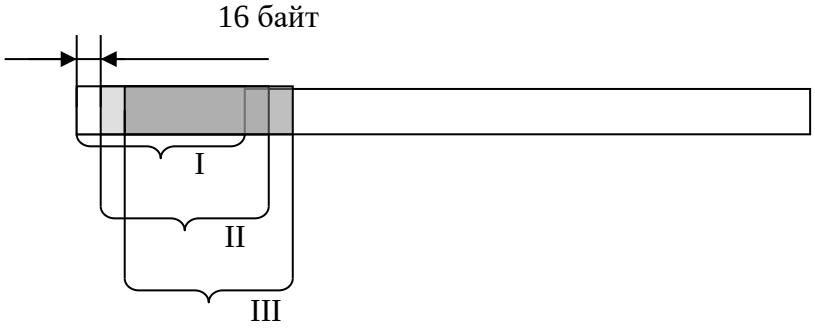


Рисунок 1.1 - Взаимное расположение сегментов памяти

Обычно программа для операционной системы MS-DOS состоит из трех сегментов: кода, данных и стека. Сегмент кода содержит команды программы, сегмент данных – обрабатываемые данные. Стек – это область памяти, которая служит для хранения информации, необходимой для вызова подпрограмм. Стек также иногда применяется для временного хранения данных. Он находится в сегменте стека. Размер сегментов кода, данных и стека не должен превышать 65536 байт для каждого сегмента.

Адрес операнда формируется как сумма адреса начала сегмента и смещения операнда внутри сегмента (рисунок 1.2).

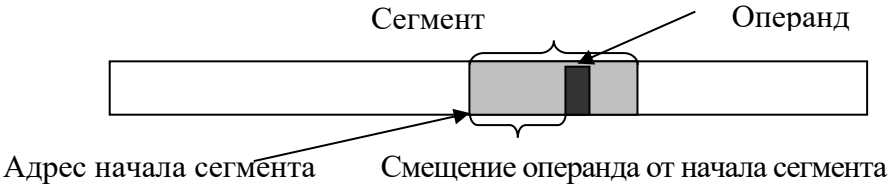


Рисунок 1.2 - Общая схема базирования адресов

Поскольку начала соседних сегментов отстоят друг от друга на 16 байт, адрес начала сегмента равен номеру сегмента, умноженному на 16. Записывают адрес обычно так: Сегмент:Смещение. Например, видеопамять текстового режима начинается с адреса 0B800h:0000, где суффикс h означает, что число шестнадцатеричное.

Операнд в МП 8086 может быть одно- и двухбайтовый. Пусть двухбайтовый операнд хранится в памяти по адресу A. В этом случае младший байт операнда

(содержащий младшие биты) находится по адресу A, а старший – по адресу A+1. Например, число 7F88h в памяти будет располагаться так (рисунок 1.3): .

Адрес	Байт
A	88h
A+1	7Fh

Рисунок 1.3 – Расположение числа в памяти

Аналогичным образом в памяти хранятся другие данные и машинные коды команд программы.

1.2. Регистры процессора

Во время работы программы команды и данные хранятся в оперативной памяти. Для повышения скорости обработки данных их можно разместить в регистрах – ячейках памяти внутри процессора. Они имеют собственные имена и обладают очень малым временем доступа к данным. МП 8086 содержит 14 регистров: AX, BX, CX, DX, SI, DI, SP, BP, CS, DS, SS, ES, IP, FLAGS. Рассмотрим их.

Первые восемь регистров являются регистрами общего назначения (РОН) и содержат 2 байта, или 16 бит. В них могут храниться любые данные, хотя каждый из них имеет собственное назначение:

- AX – аккумулятор, регистр для выполнения арифметических операций.
- BX – базовый регистр, служит для реализации индексной и косвенной адресаций, о которых будет рассказано ниже.
- CX – регистр-счетчик, используется для реализации циклов типа for, а также при операциях сдвига.
- DX – регистр данных, используется при выполнении операций умножения и для обращения к функциям операционной системы.
- SI, DI – индексные регистры. Используются для организации косвенной адресации, а совместно с BX – индексной. Регистр SI называют регистром индекса источника, а DI – регистром индекса приемника.
- SP – указатель стека. Используется при работе со стеком, и, хотя является регистром общего назначения, в большинстве программ играет только роль указателя стека. Занесение произвольных значений в него нежелательно.
- BP – указатель базы кадра в стеке. Используется при передаче параметров в подпрограмму через стек. Такой способ передачи параметров используют языки Pascal, C, C++ и другие.

Регистры AX, BX, CX, DX могут использоваться половинами, содержащими 1 байт каждая (рисунок 1.4). Старшие половины (биты 8..15) называются соответственно AH, BH, CH, DH, а младшие (биты 0..7) – AL, BL, CL, DL.

Регистры CS, DS, ES, SS – это сегментные регистры. CS хранит начальный адрес сегмента памяти, в котором находятся команды программы (сегмента кода), DS –начальный адрес сегмента, хранящего данные, а SS – начальный адрес сегмента, в котором расположен стек. Регистр ES является вспомогательным

сегментным регистром; изменяя его, можно адресовать любой сегмент оперативной памяти.

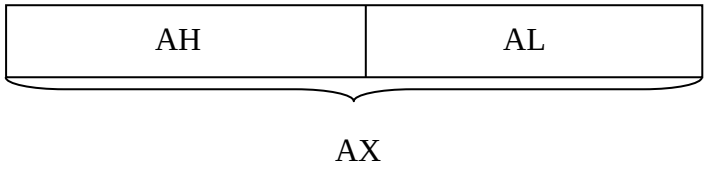


Рисунок 1.4 - Структура регистра общего назначения

Регистр IP (указатель команд) хранит смещение первого байта следующей команды внутри сегмента кода. Этот регистр изменяется автоматически при выполнении команд. Задать ему нужное значение можно с помощью команд передачи управления.

Регистр флагов FLAGS содержит информацию о результате последней арифметико-логической команды (рисунок 1.5). Команды пересылки и передачи управления на него не воздействуют. Этот регистр состоит из отдельных бит, называемых флагами, каждый из которых имеет определенный смысл. Рассмотрим некоторые из них:

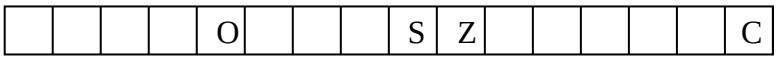


Рисунок 1.5 - Структура регистра флагов

C – флаг переноса. Равен единице, если при выполнении последней операции был перенос из старшего разряда или заем в старший разряд и нулю в противном случае. При операциях сдвига содержит последний выдвинутый из регистра бит.

Z – флаг нуля. Равен 1, если результатом последней операции был ноль. Если последней операцией была операция сравнения, единица в этом флаге указывает на то, что сравниваемые операнды равны.

S – знаковый флаг, содержит 1, если при выполнении последней операции результат был отрицательный.

O – флаг переполнения, содержит 1, если при выполнении последней операции произошел перенос в знаковый разряд (переполнение разрядной сетки).

Существуют команды условной передачи управления, которые анализируют эти биты и в зависимости от их содержимого выполняют команду, расположенную по заданному адресу, либо следующую за данной.

1.3. Способы адресации МП Intel 8086

Для адресации данных в процессоре 8086 можно использовать несколько способов. Данные хранятся в сегменте данных. Сегмент данных определяется программистом, а адреса данных задаются им с помощью меток. Пусть в сегменте

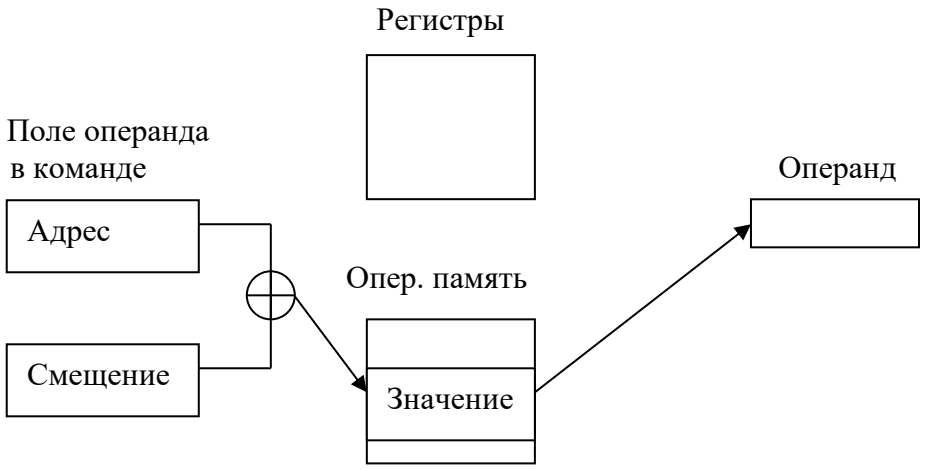
данных меткой Mas задан массив чисел каждое размером в один байт. Этой меткой мы будем пользоваться ниже при рассмотрении способов адресации со смещением.

Непосредственная адресация

Это самый простой способ адресации – вместо адреса в поле операнда команды указывается сам операнд, например: 25, -6, 100. В общем виде операнд записывается как числовая константа.

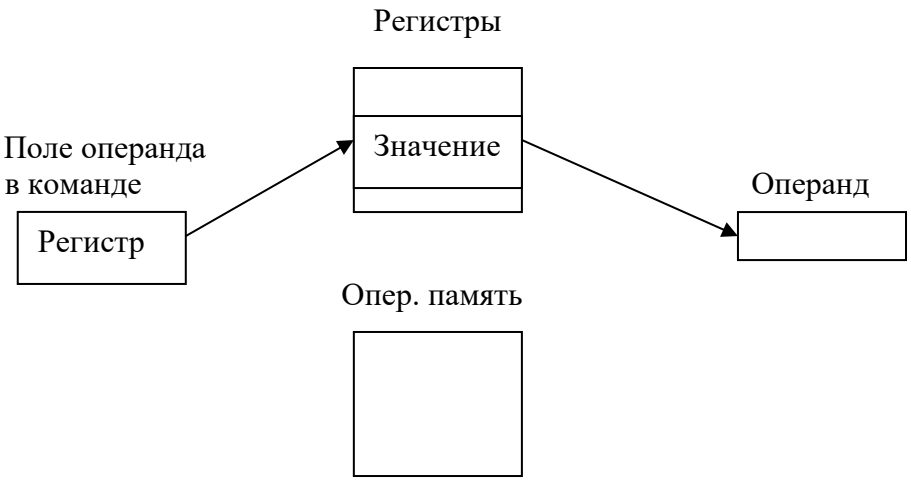
Прямая адресация

Адрес операнда задается в явном виде меткой. Адрес первого элемента массива Mas будет выглядеть так: Mas. Адрес четвертого элемента можно записать как Mas+3 (так как адрес 1-го эл-та – Mas+0, 2-го – Mas+1 и т.д.). В общем виде адрес записывается как <метка>±<константа>.



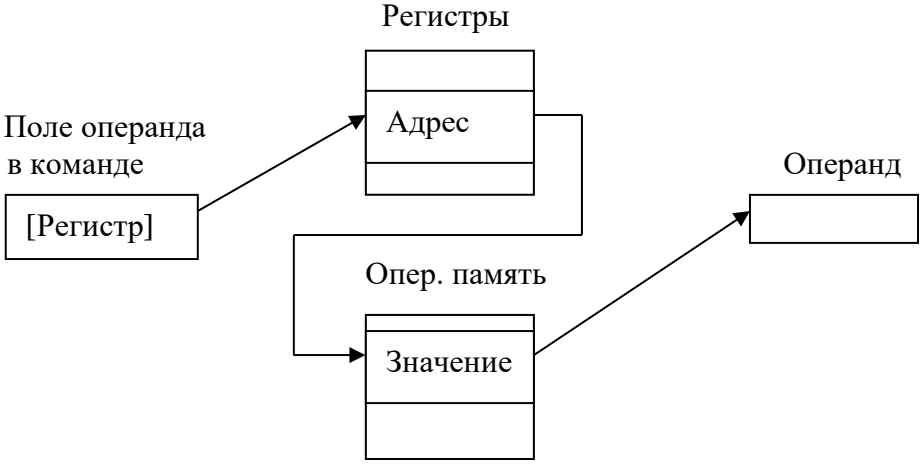
Регистровая адресация

В поле операнда указывается имя регистра. Примеры: AX, DL, SP. Общий вид: <имя регистра>.



Косвенная адресация

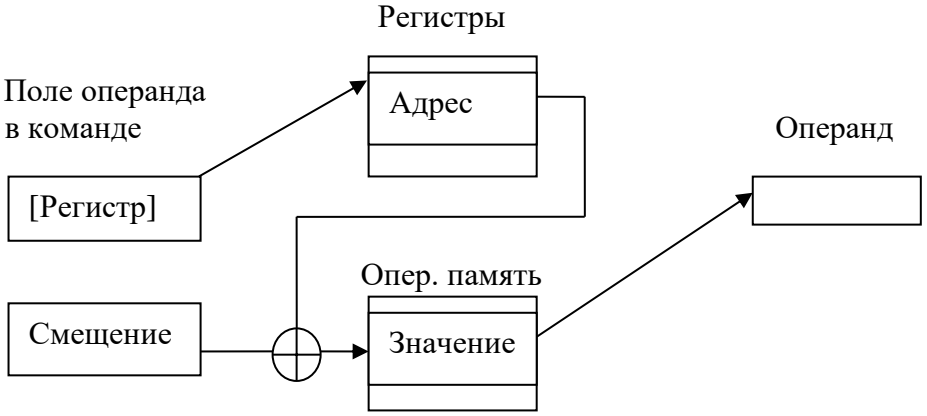
В случае косвенной адресации адрес смещение операнда в сегменте данных находится в одном из регистров BX, SI, DI. Другие регистры при косвенной адресации недопустимы. Имя регистра заключается в квадратные скобки. Примеры: [SI], [DI], [BX]. Если необходимо обратиться к данным из другого сегмента, например ES, необходимо использовать так называемую замену сегмента. Для этого перед [Рег] указывают имя сегментного регистра и ставят двоеточие, например так: ES:[DI]. Общий вид: {<сегмент>:}[<регистр>]. Фигурные скобки означают, что замена сегмента не обязательна.



Косвенная адресация со смещением

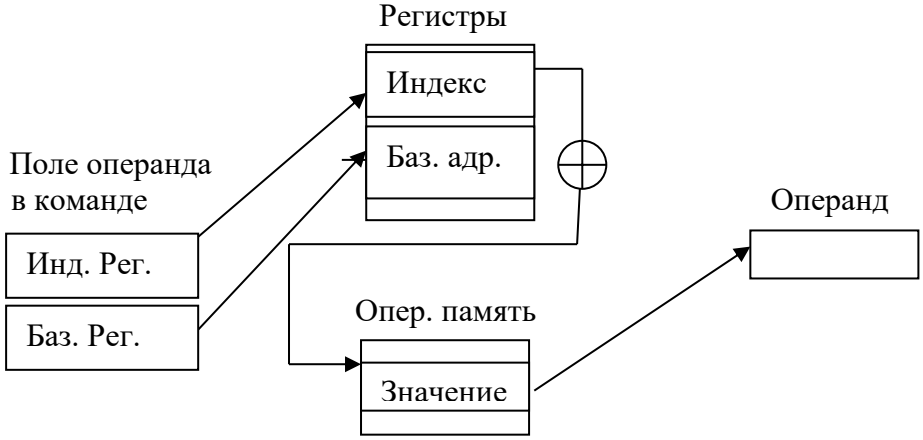
К смещению, полученному с помощью косвенной адресации, прибавляется дополнительное смещение, и по этому новому адресу и находится в оперативной памяти операнд. Существует несколько форм записи этой адресации, приведем две из них: Mas[SI] (эквивалентно [SI+смещение Mas]), [SI+4]. Смещение Mas можно найти с помощью директивы OFFSET Mas, то есть Mas[SI]≡[SI+OFFSET Mas]. В этом способе адресации можно применять еще и регистр BP, например: [BP+16]. При использовании этого регистра процессор воспринимает смещение как смещение в сегменте стека. Пример: DS:[BP-12].

Общий вид: {<сегмент>:}[<Регистр>+<Смещение>].



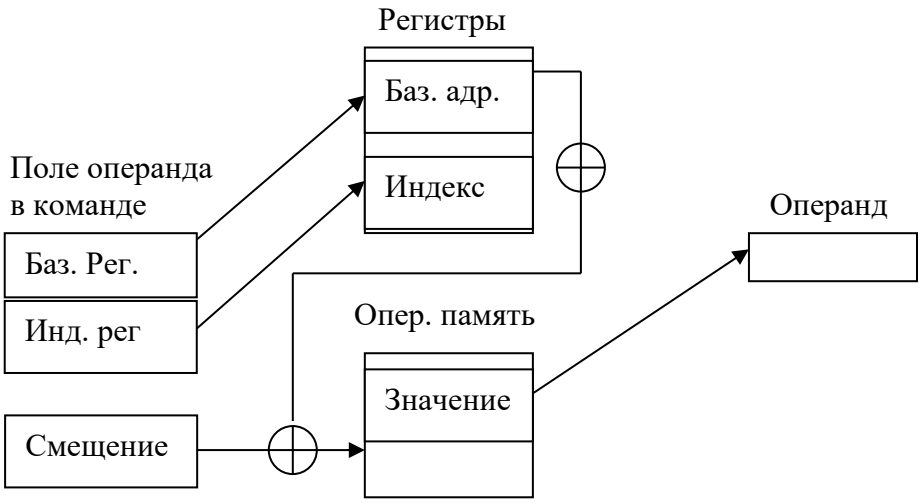
Индексная адресация

Индексная адресация часто применяется для обработки двумерных массивов. Смещение операнда вычисляется процессором как сумма содержимого базового и индексного регистра. При этом базовыми являются регистры BX, BP, а индексными SI, DI. Применение других регистров недопустимо. Примеры такой адресации: [BX][SI], [BP][DI]. Индексная адресация с базовым регистром BX обращается в сегмент данных, а с BP – в стек. Возможна замена сегмента, она выполняется аналогично косвенной адресации. Рассмотрим методику работы с двумерным массивом. В BX (или BP) заносится адрес начала строки массива, а в SI (DI) – смещение операнда внутри строки. Общий вид: {<сегмент>:}[<Баз. рег>][<Инд. регистр>].



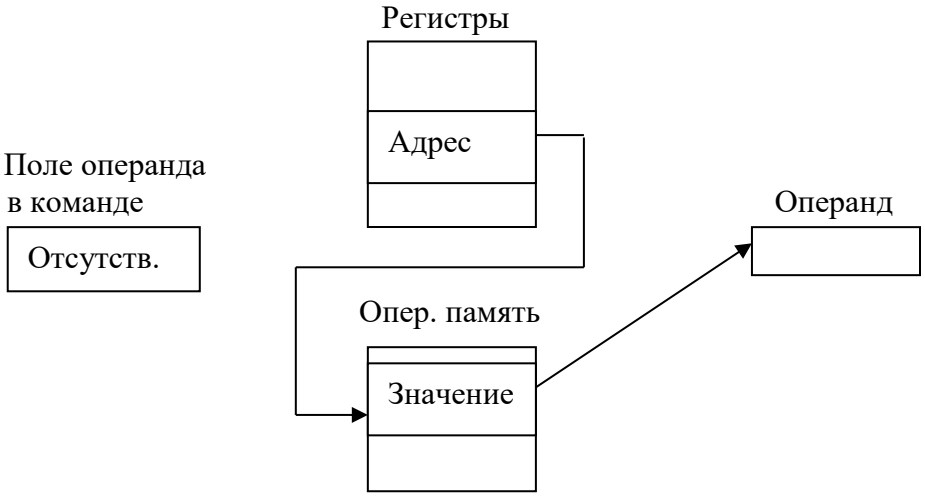
Индексная адресация со смещением

Эта адресация более удобна для обработки двумерных массивов. К смещению, полученному с помощью индексной адресации, добавляется дополнительное смещение. По аналогии с косвенной адресацией со смещением, ее можно записать как {<сегмент>:}[<Баз. рег>][<Инд. регистр> + <смещение>]. Примеры: Mas[BX][SI] ≡ [BX][SI + OFFSET Mas], [BP][SI+5], [BX][DI-7]. В этом случае в BX, BP указывают смещение строки внутри массива, а в SI, DI – смещение элемента массива в строке (индекс элемента). Также возможна замена сегмента.



Стековая адресация

Существуют команды, работающие со стеком. Они используют так называемую стековую адресацию, при которой адрес формируется из регистров SS и SP следующим образом: SS*16+SP (или SS:SP).



Неявная адресация

Существует класс команд, работающих только с одним регистром или даже одним битом одного регистра. Все они используют неявную адресацию операнда, то есть данный операнд в команде не указывается. Поскольку операнд находится в фиксированном месте, обращение к нему производится и без указания его адреса.

1.4. Указание размера операнда

Числа в памяти ЭВМ хранятся в двоичном виде в следующем формате: самый младший байт, следующий за ним байт, …, старший байт. Восьмибитовая константа может быть представлена как одним байтом, так и двумя.

Если в команде одним из операндов присутствует регистр, то размер данных в памяти и констант определяется по размеру регистра. Но в командах, проводящих арифметические операции с памятью и константой, необходимо явно указать, сколько байт занимает значение в памяти. Делается это с помощью конструкций BYTE PTR, WORD PTR и DWORD PTR. Первая устанавливает размер операнда, равный 1 байту, вторая – 2 байта, а третья – 4 байтам. Пример: WORD PTR ES:[DI] – указывает процессору, что следует использовать при выполнении операции двухбайтовое число, расположенное по адресу ES:DI. Если операнд в памяти задан меткой в сегменте данных, размер операнда уже известен ассемблеру, и его можно не указывать.

2. СТРУКТУРА ПРОГРАММЫ НА ЯЗЫКЕ АССЕМБЛЕРА

2.1. Основные понятия языка ассемблера

`Ассемблер` – это машинно-ориентированный язык низкого уровня. Программа-ассемблер заменяет мнемонические обозначения команд и операндов соответственно на коды команд и адреса операндов. Этот процесс называют ассемблированием кода.

Ассемблерная программа состоит из операторов и директив. Операторы – это инструкции, исполняемые процессором (например, MOV, ADD и т. д.). Директивы, как правило, служат для указания режимов работы ассемблера (например, директива .MODEL, см. ниже), для разбиения потока операторов на сегменты и процедуры, определения данных (также см. ниже), указания размера операнда (BYTE PTR, WORD PTR) и выполнения некоторых других операций.

Отдельные операторы или данные могут быть снабжены меткой, указывающей адрес команды либо операнда. Метка должна начинаться с латинской буквы либо символов _, @, $ и может содержать в себе латинские буквы, символы _, @, $ и цифры. В качестве метки нельзя использовать названия команд и директив.

Оператор ассемблера имеет вид:

[Метка:] Код_операции [Операнды] [; Комментарий]

В квадратных скобках находятся необязательные поля. Приведем несколько примеров:

```
Metka1      MOV  AX, ES:[DI]
            PUSH AX          ; Запись AX в стек.
            CLC              ; Сброс флага переноса
```

Программа обычно начинается с директивы

.MODEL SMALL

Она сообщает ассемблеру, что используются сегменты размером не более 64 килобайт.

Как уже было указано, программа состоит из трех сегментов: кода, данных и стека. Сегменты определяются с помощью директив SEGMENT и ENDS следующим образом:

```
Имя SEGMENT Тип
        Операторы
                …
Имя ENDS
```

где Имя – метка сегмента;

Тип – ‘code’ – сегмент кода;

‘data’ – сегмент данных;

STACK ‘stack’ – сегмент стека.

Написание сегментов будет рассмотрено ниже.

Заканчивается программа директивой END. Она имеет вид:

END метка,

где метка – адрес оператора программы, который должен быть выполнен первым при старте программы.

2.2. Написание сегмента данных

Сегмент данных предназначен для хранения данных программы таких, как глобальные переменные. Место под переменные отводится директивами определения данных. Рассмотрим некоторые из них:

DB – резервирование одного байта;

DW – резервирование одного слова из двух байт.

DD – резервирование двойного слова из 4 байт.

Эти директивы имеют 3 формы:

1) D* константа – выделение памяти под переменную и занесение в нее указанной константы

2) D* константа, константа, …, константа – выделение памяти под массив с одновременным занесением в него данных. Используется для небольших массивов

3) D* число элементов DUP(константа) – выделение памяти под массив с нужным числом элементов и занесение в каждый элемент константы. Используется для больших массивов.

Вместо константы в этих директивах можно указать символ ?, обозначающий то, что место под переменную или массив выделяется, но значение будет не определено.

Пример сегмента данных:

```
Dseg  SEGMENT  'data'
      X    DB    15                ; Переменная с начальным значением 15
      Y    DW    ?                 ; Переменная с неопределенным значением
      Mas  DB    40   DUP(0)       ; Массив из 40 элементов с нулевым
значением
      Mas2 DW    15,45,-17,0,14,1  ; Массив из 6 элементов с заданными
значениями
Dseg  ENDS
```

2.3. Написание сегмента стека

Сегмент стека обычно состоит из директивы DB 256 DUP (?), предписывающей ассемблеру выделить 256 байт под стек.

Пример сегмента стека:

```
Sseg  SEGMENT  STACK  'stack'
      DB   256   DUP  (?)
Sseg  ENDS
```

2.4. Написание сегмента кода

Сегмент кода содержит все операторы программы, разбитые на подпрограммы. В начале этого сегмента указывается директива ASSUME, имеющая такой вид:

ASSUME CS:метка сегм. кода, DS:метка сегм. данных, SS:метка сегм. стека

Подпрограммы определяются с помощью директив PROC и ENDP следующим образом:

```
Имя   PROC  Тип
      Операторы
      …
      RET     ; Возврат из подпрограммы в точку вызова
Имя   ENDP
```

Имя подпрограммы – это метка, указывающая смещение подпрограммы в сегменте кода. Тип может быть NEAR и FAR. Подпрограммы типа NEAR вызываются в пределах текущего сегмента, а FAR – из текущего или другого сегмента.

Как правило, в программах, не использующих библиотеки подпрограмм и имеющих один сегмент кода, все подпрограммы, кроме одной, делают NEAR. Подпрограмма, называемая главной (аналог main в языке C или фрагмента кода begin … end. в языке Pascal), всегда объявляется как FAR, так как в конце программы необходимо передать управление операционной системе с помощью команды, находящейся в другом сегменте.

Любая подпрограмма может вызывать другие подпрограммы (аналогично тому, как это делается в языках высокого уровня). Вызов подпрограммы осуществляется с помощью команды CALL:

CALL имя_подпрограммы

При вызове подпрограмм типа NEAR в стек записывается смещение следующей за CALL команды и производится переход к подпрограмме. По команде RET это значение из стека записывается в регистр IP, и программа продолжается со следующей за CALL командой. Для подпрограмм типа FAR в стек записывается как смещение, так и сегмент точки возврата. По команде RET в CS:IP записывается адрес точки возврата.

Пример сегмента кода:

```
Cseg   SEGMENT  'code'
ASSUME cs:Cseg, ds:Dseg, ss:Sseg
Sub    PROC NEAR
       …
       RET
Sub    ENDP

Main   PROC FAR
       …
       CALL Sub
       …
       RET
Main   ENDP
Cseg   ENDS
```

2.5. Написание головной подпрограммы

Главная подпрограмма вызывается системой, однако в стек адрес возврата не записывается. Чтобы вернуться в операционную систему, нужно выполнить команду, находящуюся в самом начале так называемого программного префикса. Программным префиксом (PSP) называется область памяти размером 256 байт, используемая для хранения переменных окружения, командной строки, некоторых других данных и предшествующая в памяти сегменту кода:

Номер сегмента PSP система записывает в регистр DS. Смещение первой его команды всегда равно нулю.

В начале работы главной подпрограммы необходимо записать в стек адрес первой команды PSP: сначала DS, а затем ноль. Тогда команда RET в конце головной подпрограммы извлечет этот адрес из стека и следующей выполнится команда возврата в операционную систему.

Поскольку при старте программы в DS находится номер сегмента PSP, в DS необходимо явно записать номер сегмента данных. Это делают так:

```
MOV  AX, Dseg
MOV  DS, AX
```

Поэтому головную процедуру пишут обычно так:

```
Main   PROC FAR
       PUSH DS          ; Занесение DS в стек
       MOV  AX, 0       ; Обнуление AX
       PUSH AX          ; Занесение нуля в стек
       MOV  AX, Dseg ; Настройка DS на сегмент данных
```

```
        MOV  DS, AX
        …
        RET
Main    ENDP
```

Пример программы на ассемблере.

Приведем пример программы, вычисляющей сумму трех чисел: X,Y,Z и записывающей результат в переменную RES.

```
        .MODEL      SMALL
        ; Сегмент стека
Sseg    SEGMENT     STACK       ‘stack’
        DB      256     DUP     (?)
Sseg    ENDS
        ; Сегмент данных
Dseg    SEGMENT     ‘data’
        X       DB      3
        Y       DB      5
        Z       DB      7
        RES     DB      ?
Dseg    ENDS
        ; Сегмент кода
Cseg    SEGMENT     ‘code’
ASSUME      CS:Cseg, DS:Dseg, SS:Sseg
        ; Процедура добавления Z к сумме X и Y в AL и записи результата в RES
Sumres PROC NEAR
        ADD  AL, Z
        MOV  RES, AL
        RET
Sumres ENDP
        ; Головная подпрограмма
Main    PROC FAR
        ; Подготовимся к возврату в операционную систему
        PUSH DS
        MOV  AX, 0
        PUSH AX
        ; Настроим DS на наш сегмент данных
        MOV  AX, Dseg
        MOV  DS, AX
        ; Делаем вычисления
        MOV  AL, X
        ADD  AL, Y
        CALL Sumres
        ; Передача управления первому оператору в PSP
        RET
Main    ENDP
Cseg    ENDS
        END   Main
```

2.6. Подготовка и отладка программ в Emu8086

Emu8086 сочетает в себе редактор исходного кода, ассемблер, дизассемблер и программный эмулятор (виртуальный ПК) с отладчиком. Эта программа компилирует исходный код и выполняет его с помощью эмулятора шаг за шагом.

Визуальный интерфейс программы прост в работе, он позволяет наблюдать регистры, флаги и память во время выполнения программы.

Арифметико-логическое устройство (АЛУ) показывает внутреннюю работу центрального процессора.

Эмулятор выполняет программы на виртуальном ПК, который полностью исключает возможность доступа из программы к реальным аппаратным средствам, таким как жесткие диски и память. При этом отладка становится более легкой.

При открытии программы появляется окно (рисунок 2.1):

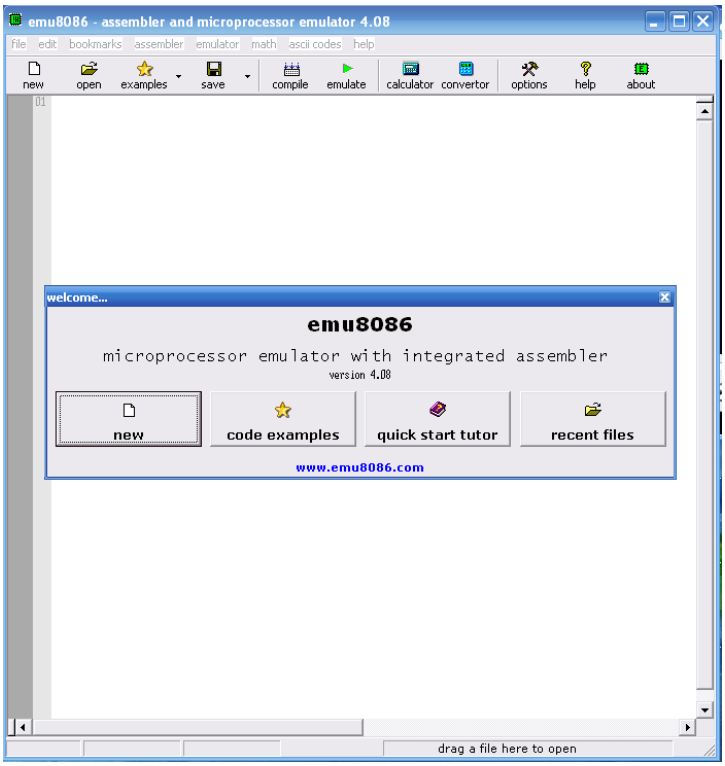


Рисунок 2.1 - Открытие программы emu8086

При создании нового исполнимого файла с расширением EXE открывается окно редактора исходного кода (рисунок 2.2), в котором необходимо написать текст программы.

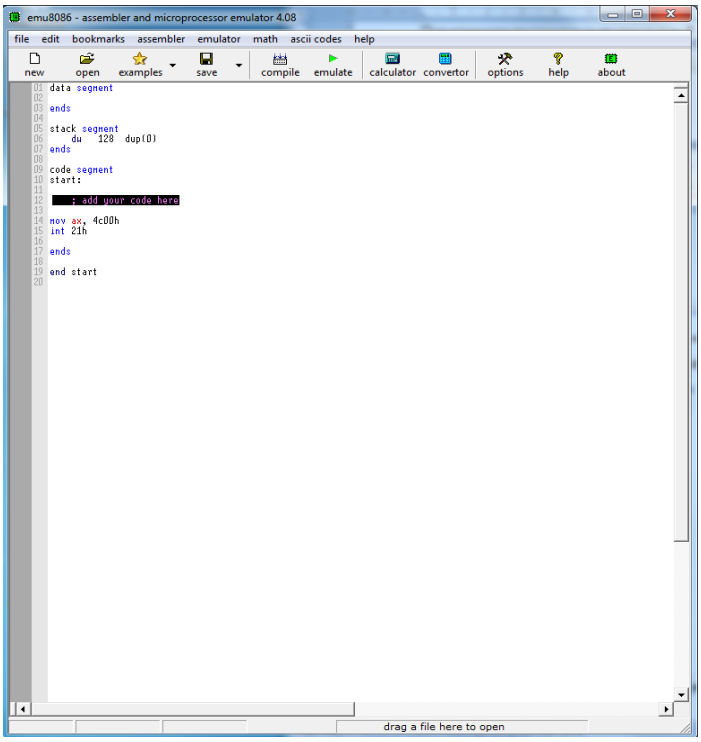


Рисунок 2.2 – Редактор кода

Для выполнения программы надо нажать кнопку **EMULATE** или **F5**. При этом откроются два окна: эмулятора и окно исходного кода (рисунок 2.3).

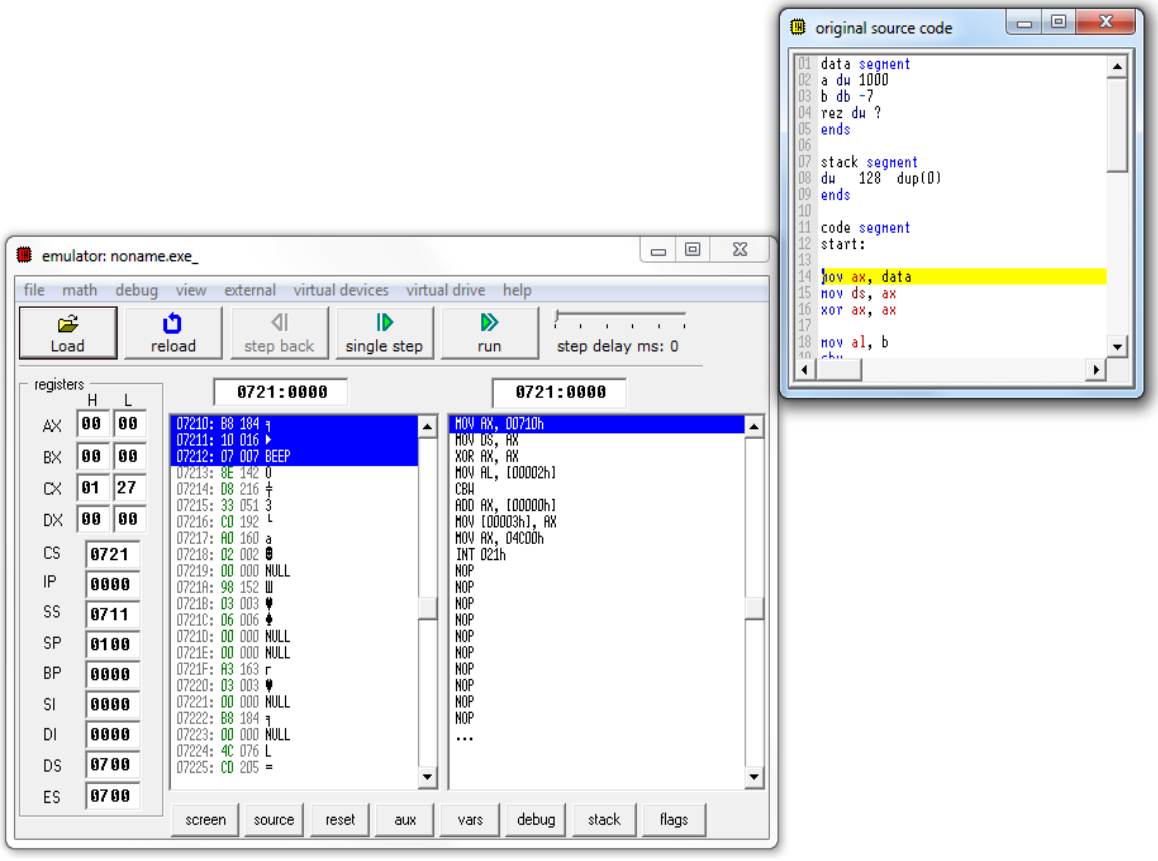


Рисунок 2.3 - Окно эмулятора и окно исходного кода

В окне эмулятора отображаются регистры и находятся кнопки управления программой. В окне исходного кода отображается исходный текст программы, где подсвечивается строка, которая выполняется в данный момент. В окне эмулятора можно запустить программу на выполнения целиком (**RUN**) либо в пошаговом режиме (**SINGLE STEP**). Пошаговый режим удобен для отладки.

В соответствующих окнах также можно просматривать содержимое сегмента данных, регистра флагов и стека.

3. ЛАБОРАТОРНЫЙ ПРАКТИКУМ

Лабораторная работа №1

ПРЕДСТАВЛЕНИЕ ДАННЫХ. АРИФМЕТИКО-ЛОГИЧЕСКИЕ ОПЕРАЦИИ

`Цель работы`: изучение архитектуры МП Intel 8086, изучение структуры простейшей ассемблерной программы, ознакомление с системой арифметико-логических команд процессора, организация вычислений на языке ассемблера.

Методические указания

При выполнении арифметико-логических команд наибольшего быстродействия и удобства программирования можно достичь за счет использования аккумулятора (регистра AX) для хранения промежуточных результатов. Например, вычисление выражения D=A+B-C можно записать так:

```
MOV  AX,A
ADD  AX,B
SUB  AX,C
MOV  D,AX
```

При реализации операций деления необходимо помнить о том, что если результат не помещается целиком в регистре-приемнике (например, при делении 8B00h в регистре AX на 3 в регистре BL), возникает ошибка «деление на ноль» и программа аварийно завершается. Чтобы избежать подобных ситуаций, следует увеличивать размерность делимого и делителя. В нашем примере следует командой CWD расширить разрядность делимого и делить на BX, а не на BL (естественно, при этом BH должен быть равен 0).

Деление и умножение на степень двойки следует выполнять с помощью команд сдвига. Эти команды наиболее эффективны при их выполнении для регистра AX.

Практическая часть

Практическая часть работы включает выполнение следующих действий:

– формирование числовых значений в соответствии с индивидуальным заданием, определение минимального формата представления исходных данных;

– по заданному алгоритму составление и выполнение программы работы с данными.

Правильность разработки и выполнения программ арифметико-логической обработки данных контролируется путем ручной трассировки заданных алгоритмов с последующим сравнением результатов работы программ с результатами ручной трассировки.

Варианты заданий

Значения исходных данных, которые должны храниться в сегменте данных, определяются выражениями:

$X1=№B*(-1)^{№B}$

$X2=(-1)^{№B+1}*(№Г*№B)$

$X3=(-1)^{№B+2}*(№Г*№B+№Г)$

$X4=(-1)^{№B+3}*№Г$

где №В – номер варианта, №Г – номер группы.

Варианты алгоритмов программ приведены на рисунках 3.1, 3.2.

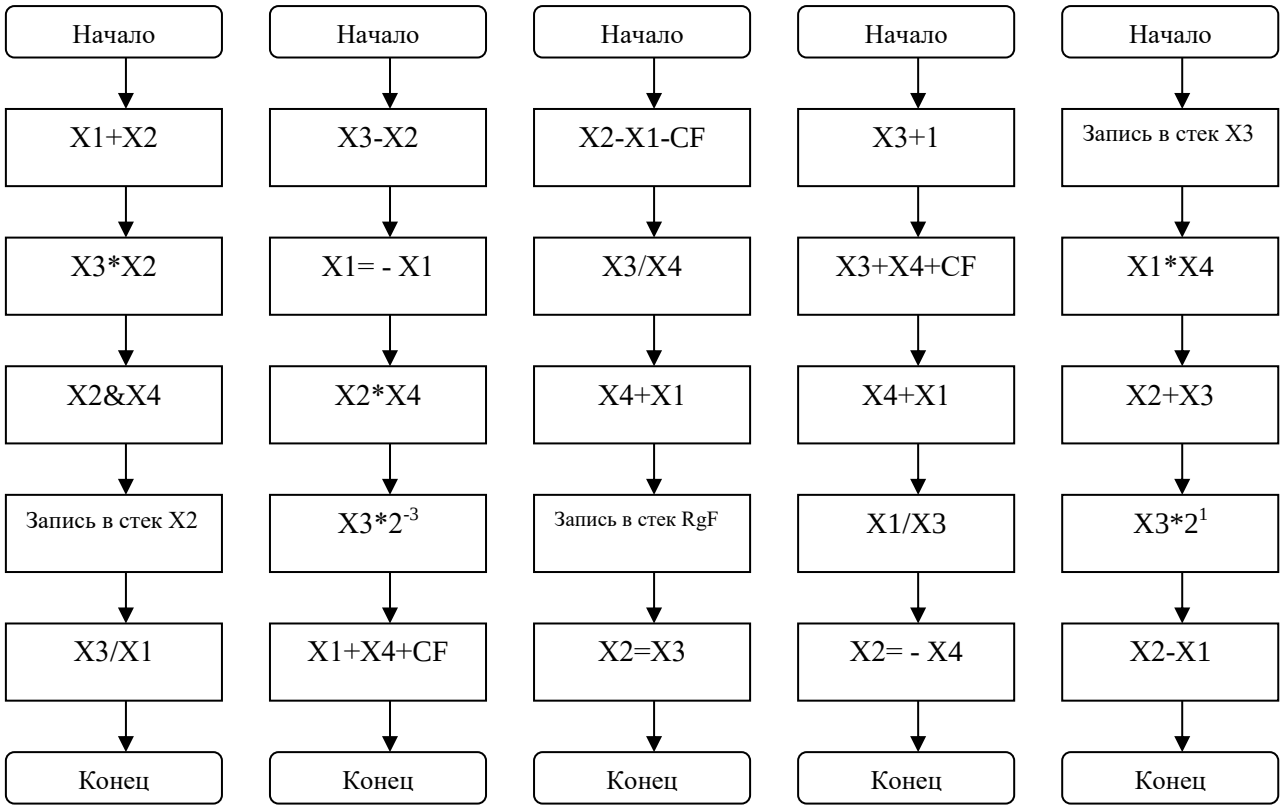


Рисунок 3 .1 - Алгоритмы программ

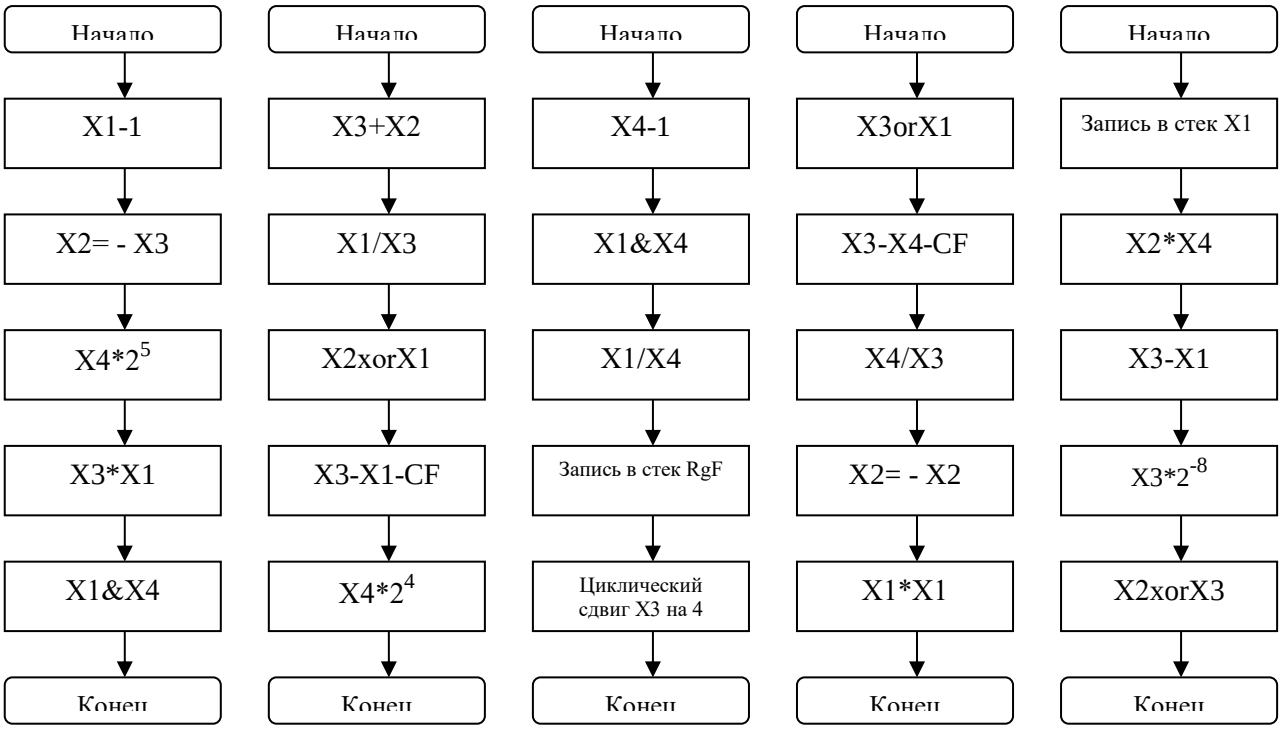


Рисунок 3 .2. -Алгоритмы программ

Порядок выполнения работы

1. Определить исходные данные в соответствии с номером варианта.
2. Перевести значения величин X1-X4 в шестнадцатеричную систему счисления.
3. Провести трассировку заданного алгоритма с использованием заданных исходных данных.
4. Составить программу заданного алгоритма в мнемокодах.
5. Оформить отчет по лабораторной работе.
6. В учебной лаборатории проверить результаты выполнения программы в программе-отладчике, сравнивая их с результатами ручной трассировки алгоритма.

Содержание отчета

1. Титульный лист.
2. Текст задания.
3. Алгоритм программы.
4. Текст программы на ассемблере с комментариями.
5. Таблица трассировки программы.

Лабораторная работа №2

УСЛОВНЫЕ И БЕЗУСЛОВНЫЕ ПЕРЕХОДЫ

Цель работы: изучение команд условного перехода, организации условных операторов и итеративных циклов.

Методические указания

Условный оператор с одноальтернативным ветвлением организуется в языке ассемблера следующим образом:

```
        Jусловие End_if
        …
        Операторы
        …
        End_if:
```

Заметим, что операторы в данном случае выполняются при *невыполнении* условия, в отличие от языков высокого уровня. Например, оператор языка Pascal

```
If A=0 then
        A=5;
```

на языке ассемблера будет выглядеть так:

```
                CMP   WORD PTR A, 0
                JNZ    End_if1
                MOV  WORD PTR A, 5
End_if1         …
```

Как видно, в операторе If языка ассемблера условие заменяется на противоположное тому, что использовалось бы в языке высокого уровня.

Двухальтернативный условный оператор записывается в ассемблере так:

```
        Jусловие      If_ops
        …
        Операторы ветви Else
        …
        JMP End_if
If_ops:         …
        Операторы ветви If
        …
End_if:
```

Здесь условие менять на противоположное не надо, так как при его выполнении выполнится блок If, а при невыполнении – блок Else.

Циклы с предусловием записываются в языке ассемблера следующим образом:

```
Cycle1:         Jусловие      End_cycle1
                …
                Операторы
                …
                JMP    Cycle1
End_cycle1:  …
```

Условие, как и для одноальтернативных условных операторов, нужно изменить по сравнению с языками высокого уровня на противоположное.

Пример: цикл языка Pascal

```
While A<>0 do
        A=A shr 1;
```

на ассемблере запишется так:

```
Cycle1:      CMP  BYTE PTR A, 0
             JZ   End_cycle1
             MOV  AL,A
             SAR  AL,1
             MOV  A,AL
             JMP  Cycle1
End_cycle1:  …
```

Циклы с постусловием записываются на ассемблере так:

```
Cycle1:      …
             Тело цикла
             …
             Jусловие    Cycle1
```

Для циклов языка Pascal условие необходимо изменить на противоположное. Что касается языка С, то в нем используется именно такая конструкция цикла с постусловием.

Практическая часть

Практическая часть работы включает выполнение следующих действий:

– формирование исходных числовых значений;

– в соответствии с индивидуальным заданием составление алгоритма программы для решения поставленной задачи;

– по алгоритму составление и выполнение программы.

Правильность разработки и выполнения контролируется путем ручной трассировки составленного алгоритма с последующим сравнением результатов работы программы с результатами ручной трассировки.

Варианты заданий

Для всех заданий исходное число (числа) хранится в двухбайтовой ячейке (ячейках) сегмента данных, результат необходимо сохранить в однобайтовую ячейку сегмента данных. Под словосочетанием «сохранить результат» понимается запись результата в однобайтовую ячейку в сегменте данных. Во всех заданиях следует использовать только итерационные циклы и условные операторы.

1. Подсчитать вес двоичного вектора и сохранить результат.
2. Если число в состоит менее, чем из 3 десятичных цифр, сохранить их сумму, иначе сохранить 0.
3. Найти и сохранить сумму четных десятичных цифр заданного числа.
4. Найти и сохранить сумму нечетных десятичных цифр заданного числа.
5. Найти и сохранить количество десятичных цифр в числе.
6. Найти максимальную цифру в числе и сохранить ее.
7. Найти и сохранить минимальную цифру в числе.

8. Найти и сохранить номер максимальной цифры в числе, считая, что младшая цифра имеет номер 1, и номера увеличиваются в сторону старших цифр.
9. В условиях задания №8 найти и сохранить номер минимальной цифры числа.
10. Сохранить 1, если число содержит данную цифру, иначе сохранить 0.
11. Найти и сохранить индекс первой со стороны младших цифр четной цифры числа, учитывая, что индекс вычисляется по правилам из задания №8.
12. Найти сумму первых N натуральных чисел и сохранить ее.
13. Найти сумму первых N натуральных положительных чисел и сохранить ее.
14. Найти сумму первых N натуральных четных чисел и сохранить ее.
15. Найти сумму первых N натуральных нечетных чисел и сохранить ее.

Порядок выполнения работы

1. Сформировать исходные данные в соответствии с вариантом.
2. Составить алгоритм программы.
3. Провести трассировку заданного алгоритма с использованием заданных исходных данных.
4. Составить программу заданного алгоритма в мнемокодах.
5. Оформить отчет по лабораторной работе.
6. В учебной лаборатории проверить результаты выполнения программы в программе-отладчике, сравнивая их с результатами ручной трассировки алгоритма.

Содержание отчета

1. Титульный лист.
2. Текст задания.
3. Алгоритм программы.
4. Текст программы на ассемблере с комментариями.
5. Таблица трассировки программы.

Лабораторная работа №3

ОБРАБОТКА МАССИВОВ

Цель работы: изучение работы с массивами, организации арифметических циклов в языке ассемблера.

Методические указания

Работа с массивами возможна при использовании нескольких способов адресации (см. раздел 1.3): косвенной и индексной (как со смещением, так и без него).

Арифметические циклы на языке ассемблера организуются следующим образом:

```
        MOV  CX, число_итераций
Cycle1:      …
```

```
        Тело цикла
        …
        LOOP Cycle1
```

Следует помнить, что данный цикл в ассемблере всегда имеет форму:

```
For cx:=число_итераций downto 1 do
        Тело_цикла;
```

Если необходимо использовать другой цикл, например, for i:=1 to число_итераций do тело_цикла, нужно дополнительно использовать ячейку памяти или регистр для использования в роли i. *Пример*:

```
        MOV  SI,1
        MOV  CX, число_итераций
Cycle1:         …
        Тело цикла
        …
        INC    SI
        LOOP Cycle1
```

Если необходим шаг цикла, отличный от единицы, следует вместо INC SI использовать ADD SI, шаг_цикла.

Практическая часть

Практическая часть работы включает выполнение следующих действий:

- в соответствии с индивидуальным заданием составление алгоритма программы обработки массивов, содержащих 10 элементов;
- по алгоритму составление и выполнение программы;
- контроль результатов работы программы.

Правильность разработки и выполнения контролируется путем ручной трассировки составленного алгоритма с последующим сравнением результатов работы программы с результатами ручной трассировки.

Варианты заданий

Для всех заданий исходный массив хранится в сегменте данных, результаты необходимо сохранить в РОНы.

1. Найти логическую сумму положительных элементов массива и записать ее в Rg AX, и логическую сумму отрицательных элементов массива, записать ее в Rg BX (формат элементов массива - байт).
2. Найти максимальный элемент массива и записать его в Rg BH (формат элементов массива - байт).
3. Найти сумму элементов массива, значение которых ≥ 3, и записать ее в Rg AL (формат элементов массива - байт).
4. Посчитать количество элементов массива, равных нулю, и записать их в Rg AX (формат элементов массива - слово).
5. Найти результат умножения максимального элемента массива на 25 и записать его в Rg BX, Rg CX (формат элементов массива - слово).
6. Найти результат деления числа 100 на минимальный элемент массива и записать в Rg BX (формат элементов массива - байт).

7. Найти количество положительных, нулевых и отрицательных элементов массива и записать в Rg AL, Rg BL и Rg DL соответственно (формат элементов массива - байт).
8. Найти минимальный положительный элемент массива (его номер и значение) и записать в Rg BX и Rg DX соответственно (формат элементов массива - слово).
9. Найти арифметическую сумму элементов массива, значения которых лежат в интервале $-10 < X(i) < 20$, и записать ее в Rg DH (формат элементов массива - байт).
10. Найти количество элементов массива, имеющих отрицательное значение и четный номер, и записать в Rg AX (формат элементов массива - слово).
11. Найти элемент массива, имеющий максимальное абсолютное значение, и записать в Rg CX (формат элементов массива - байт).
12. Найти отрицательный элемент массива, имеющий максимальное абсолютное значение, и записать в Rg DX (формат элементов массива - слово).
13. Найти количество элементов массива, значения которых лежат в интервале $-20<X(i)< 50$, и записать в Rg BX (формат элементов массива - слово).
14. Найти сумму модулей элементов массива и записать в Rg DX (формат элементов массива - байт).
15. Найти результат умножения индекса минимального элемента на 55 и записать его в Rg BX (формат элементов массива - байт).

Порядок выполнения работы

1. Сформировать исходные данные в соответствии с вариантом.
2. Составить алгоритм программы для решения поставленной задачи.
3. Провести трассировку заданного алгоритма с использова-нием заданных исходных данных.
4. Составить программу заданного алгоритма в мнемокодах.
5. Оформить отчет по лабораторной работе.
6. В учебной лаборатории проверить результаты выполнения программы в программе-отладчике, сравнивая их с результатами ручной трассировки алгоритма.

Содержание отчета

1. Титульный лист.
2. Текст задания.
3. Алгоритм программы.
4. Текст программы на ассемблере с комментариями.
5. Таблица трассировки программы.

Лабораторная работа №4

ОРГАНИЗАЦИЯ РАБОТЫ С ПОДПРОГРАММАМИ

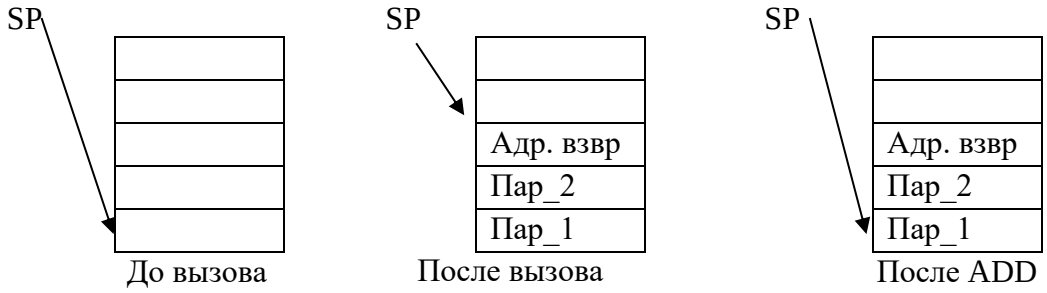
Цель работы: изучение организации вызова подпрограмм в языке ассемблера, передачи параметров через стек и возврат значений из функции.

Методические указания

Передача параметров в подпрограммы языков высокого уровня обычно производится через стек. Поэтому вызов подпрограммы выглядит обычно следующим образом:

```
PUSH пар_1
…
PUSH пар_N
CALL subrtn
ADD  sp, N*2
```

Рассмотрим, что происходит со стеком:



Стандартная подпрограмма с параметрами, передаваемыми через стек, на языке ассемблера выглядит так:

```
Subrtn    PROC     NEAR
          PUSH BP  ; Сохранить старое значение BP
          MOV  BP, SP
          …
          Тело подпрограммы
          …
          POP  BP  ; Восстановить значение BP
Subrtn    ENDP
```

Рассмотрим стек после выполнения первых двух операторов подпрограммы:

BP	BP=SP
Адр. взвр.	BP+2
Пар_2	BP+4
Пар_1	BP+6

Видно, что первый параметр находится по адресу SS:[BP+6], а второй – по адресу SS:[BP+4].

Возврат значения в функциях языков высокого уровня осуществляется через регистр AX (AL, DX:AX), в зависимости от размера возвращаемого значения.

Приведем пример функции, складывающей значения параметров и возвращающей результат через AX:

```
Sum   PROC NEAR
            PUSH BP
            MOV  BP, SP
            MOV  AX, SS:[BP+6]
            ADD  AX, SS:[BP+4]
            POP  BP
            RET
Sum   ENDP
```

Заметим, что все, что было записано в стек внутри подпрограммы, должно быть извлечено из него, так как в противном случае команда RET возвратит управление не в ту точку, откуда была вызвана подпрограмма.

Практическая часть

Практическая часть работы включает выполнение следующих действий:

– в соответствии с индивидуальным заданием составление алгоритма основной программы и подпрограммы;

– по алгоритму составление и выполнение программы;

– контроль результатов работы программы.

Правильность разработки и выполнения контролируется путем ручной трассировки составленного алгоритма с последующим сравнением результатов работы программы с результатами ручной трассировки.

Варианты заданий

Для всех заданий входные данные передаются в подпрограмму через стек, а результат возвращается через регистр AL. Для массивов входными данными являются адрес массива и число элементов в нем.

1. Найти НОД 2 заданных чисел.
2. Найти максимум в заданном массиве.
3. Найти минимум в заданном массиве.
4. Найти сумму четных элементов массива.
5. Найти сумму нечетных элементов массива.
6. Подсчитать число ненулевых элементов массива.
7. Найти индекс минимального элемента в массиве.
8. Найти индекс максимального элемента в массиве.
9. Вычислить i-е число Фибоначчи.
10. Вычислить значение функции $F(x)=x^2+5x+7$
11. Вернуть 1, если числа являются сторонами треугольника Пифагора, иначе вернуть 0.
12. Вернуть 1, если точка лежит внутри окружности и 0 иначе.

Порядок выполнения работы

1. Сформировать исходные данные в соответствии с вариантом.

2. Составить алгоритм подпрограммы и основной программы для решения поставленной задачи.
3. Провести трассировку заданного алгоритма с использованием заданных исходных данных.
4. Составить программу заданного алгоритма в мнемокодах.
5. Оформить отчет по лабораторной работе.
6. В учебной лаборатории проверить результаты выполнения программы в программе-отладчике, сравнивая их с результатами ручной трассировки алгоритма.

Содержание отчета

1. Титульный лист.
2. Текст задания.
3. Алгоритм программы.
4. Текст программы на ассемблере с комментариями.
5. Таблица трассировки программы.

Лабораторная работа №5

ИСПОЛЬЗОВАНИЕ МАКРОСРЕДСТВ

Цель работы: изучение работы с макроопределениями и макрокомандами в языке ассемблера.

Методические указания

Макросредства, как и процедуры, используются для борьбы с повторяющимися действиями.

В случае использования в программе макросредств применяется расширенный вариант языка – макроязык. Программа в этом случае транслируется в два этапа: сначала – к языку без макросредств (этим занимается макрогенератор), затем – на машинный язык (эту задачу решает ассемблер). Макрогенератор и ассемблер являются частями одного транслятора - макроассемблера.

К макросредствам относятся макросы и макрокоманды. Часто повторяющийся фрагмент программы описывается в виде макроса, а затем в нужных местах выписываются ссылки на него. Макрогенератор вместо ссылки на макрос подставляет сам макрос.

Описание макроса (макроопределение) выглядит следующим образом:

```
Имя_макроса MACRO Формальные_параметры
[Тело макроса]
ENDM
```

Макроопределение размещают в начале исходного текста программы до сегмента кода и данных с тем, чтобы не ухудшать читабельность программы.

Приведем пример:

```
SUM MACRO x, y
MOV AX, y
ADD x, AX
```

ENDM

Ссылкой на макрос является макрокоманда. Она формируется следующим образом:

Имя_макроса Фактические_параметры

Примером может быть макрокоманда

SUM a, ES:b

Число фактических параметров должно соответствовать числу формальных параметров. Лишние фактические параметры игнорируются, в качестве недостающих используются пустые тексты.

Когда макрогенератор встречает макрокоманду, то он выполняет макроподстановку: находит описание макроса с указанным именем, берет его тело, заменяет в этом теле все формальные параметры на соответствующие фактические параметры и полученный таким образом текст (он называется макрорасширением) подставляет в программу вместо макрокоманды.

Приведем пример процесса такого замещения (макрогенерации)

$$\text{SUM a, ES:b} \xrightarrow{x \to a \quad y \to ES:b} \begin{matrix}\text{MOV AX, ES:b} \\ \text{ADD a, AX}\end{matrix}$$

Практическая часть

Практическая часть работы включает выполнение следующих действий:

– в соответствии с индивидуальным заданием составление алгоритма программы с учетом использования макросредств;

– по алгоритму составление и выполнение программы;

– контроль результатов работы программы.

Правильность разработки и выполнения контролируется путем ручной трассировки составленного алгоритма с последующим сравнением результатов работы программы с результатами ручной трассировки.

Варианты заданий

Варианты заданий берутся из лабораторной работы №4.

Для всех заданий входные данные при обращении к макросу передаются через РОНы, а результат возвращается через регистр AL. Для массивов входными данными являются адрес массива и число элементов в нем.

1. Найти НОД 2 заданных чисел.
2. Найти максимум в заданном массиве.
3. Найти минимум в заданном массиве.
4. Найти сумму четных элементов массива.
5. Найти сумму нечетных элементов массива.
6. Подсчитать число ненулевых элементов массива.
7. Найти индекс минимального элемента в массиве.
8. Найти индекс максимального элемента в массиве.
9. Вычислить i-е число Фибоначчи.
10. Вычислить значение функции $F(x)=x^2+5x+7$
11. Вернуть 1, если числа являются сторонами треугольника Пифагора, иначе вернуть 0.

12. Вернуть 1, если точка лежит внутри окружности и 0 иначе.

Порядок выполнения работы

1. Сформировать исходные данные в соответствии с вариантом.
2. Составить алгоритм программы для решения поставленной задачи.
3. Провести трассировку заданного алгоритма с использованием заданных исходных данных.
4. Составить программу заданного алгоритма в мнемокодах.
5. Оформить отчет по лабораторной работе.
6. В учебной лаборатории проверить результаты выполнения программы в программе-отладчике, сравнивая их с результатами ручной трассировки алгоритма.

Содержание отчета

1. Титульный лист.
2. Текст задания.
3. Алгоритм программы.
4. Текст программы на ассемблере с комментариями.

Список используемых источников

1. [004.43 (75) Ю78] Юров В.И. Assembler: Учебник для вузов. – СПб: ПИТЕР, 2006. – 636 с. Количество экз. в библ. – 106.
2. [004.2 Т18] Таненбаум Э. Архитектура компьютера. Пер. И. Ткачева. – М. и др.: Питер, 2005. – 698 с. Количество экз. в библ. – 20.
3. [004.43 Г62] Голубь Н.Г. Искусство программирования на АССЕМБЛЕРЕ. – СПб.: Diasoft, 2006. – 820 с. Количество экз. в библ. – 5.
4. [519.682] Пильщиков В.Н. Программирование на языке ассемблера IBM PC. М.: Диалог-МИФИ, 1998. – 288 с. Количество экз. в библ. – 50.
5. [004.43 Р83] Рудаков П.И., Финогенов К.Г. Язык ассемблера: уроки программирования. – М.: Диалог-МИФИ, 2001.-640 с. Количество экз. в библ. – 131.

СОДЕРЖАНИЕ

Введение 3
1. МИКРОПРОЦЕССОР INTEL 8086 3
1.1. Общие принципы работы МП 8086 при выполнении прикладных программ . 3
1.2. Регистры процессора 5
1.3. Способы адресации МП Intel 8086 6
1.4. Указание размера операнда 11
2. СТРУКТУРА ПРОГРАММЫ НА ЯЗЫКЕ АССЕМБЛЕРА 11
2.1. Основные понятия языка ассемблера 11
2.2. Написание сегмента данных 12
2.3. Написание сегмента стека 13
2.4. Написание сегмента кода 13
2.5. Написание головной подпрограммы 14
2.6. Подготовка и отладка программ в Emu8086 16
3. ЛАБОРАТОРНЫЙ ПРАКТИКУМ 18
Список используемых источников 30
ПРИЛОЖЕНИЕ 32

ПРИЛОЖЕНИЕ

Основные команды МП 8086

Таблица П.1

Команда	Действие	Z	S	C	O
MOV Оп1, Оп2	Оп1←Оп2	-	-	-	-
XCHG Оп1, Оп2	Оп1↔ Оп2	-	-	-	-
NEG Оп1	Оп1:=-Оп1				
ADD Оп1, Оп2	Оп1:= Оп1+Оп2	*	*	*	*
ADC Оп1, Оп2	Оп1:= Оп1+Оп2+Флаг С	*	*	*	*
SUB Оп1, Оп2	Оп1:= Оп1-Оп2	*	*	*	*
SBB Оп1, Оп2	Оп1:= Оп1-Оп2-Флаг С	*	*	*	*
INC Оп	Оп1:= Оп1+1	*	*	-	*
DEC Оп	Оп1:= Оп1-1	*	*	-	*
MUL Оп, IMUL Оп	DX:AX = AX*Оп	*	-	*	*
	AX = AL*Оп	*	*	*	*
DIV Оп, IDIV Оп	AX = DX:AX div Оп; DX = DX:AX mod Оп	*	-	*	*
	AL = AX div Оп; AH = AX mod Оп	*	*	*	*
CMP Оп1, Оп2	Оп1-Оп2 без сохранения результата	*	*	*	*
NOT Оп1	Оп1:= ¬Оп1				
AND Оп1, Оп2	Оп1:= Оп1 ^ Оп2	*	*	0	0
OR Оп1, Оп2	Оп1:= Оп1∨Оп2	*	*	0	0
XOR Оп1, Оп2	Оп1:= Оп1⊕ Оп2	*	*	0	0
TEST Оп1, Оп2	Оп1 ^ Оп2 без сохранения результата	*	*	0	0
SHL Оп1,1 SHL Оп1,CL	Логический сдвиг влево на 1 или CL бит	*	*	*	*
SHR Оп1,1 SHR Оп1,CL	Логический сдвиг вправо на 1 или CL бит	*	*	*	*
SAR Оп1,1 SAR Оп1,CL	Арифметический сдвиг вправо на 1 или CL бит	*	*	*	0
JMP метка	Переход на метку	-	-	-	-
JC метка JB метка	Переход, если С=1 (если Оп1 в CMP<Оп2 и Оп1 и Оп2 - беззнаковые)	-	-	-	-
JNC метка JNB метка JAE метка	Переход, если С=0 (если Оп1 в CMP>=Оп2 и Оп1 и Оп2 - беззнаковые)	-	-	-	-
JZ метка	Переход, если Z=1 (если Оп1 в CMP=Оп2 или если результат нулевой)	-	-	-	-

Продолжение табл. П.1

Команда	Действие	Z	S	C	O
JNZ метка	Переход, если Z=0 (если Оп1 в CMP<>Оп2 или если результат ненулевой)	-	-	-	-
JS метка	Переход, если S=1 (если результат отрицательный)	-	-	-	-
JNS метка	Переход, если S=0 (если результат положительный)	-	-	-	-
JBE метка JNA метка	Переход, если C∨Z=1 (если Оп1 в CMP<=Оп2 и Оп1, Оп2 – беззнаковые)	-	-	-	-
JA метка	Переход, если C=0 и Z=0 (если Оп1>Оп2 и Оп1, Оп2 – беззнаковые)	-	-	-	-
JL метка	Переход, если O⊕S = 1 (если Оп1 в CMP<Оп2 и Оп1 и Оп2 – знаковые)	-	-	-	-
JNL метка JGE метка	Переход, если O⊕S = 0 (если Оп1 в CMP>=Оп2 и Оп1 и Оп2 – знаковые)	-	-	-	-
JLE метка JNG метка	Переход, если O⊕S = 1 и C=1 (если Оп1 в CMP<=Оп2 и Оп1 и Оп2 – знаковые)	-	-	-	-
JG метка	Переход, если O⊕S = 0 и Z=0 (если Оп1 в CMP>Оп2 и Оп1 и Оп2 – знаковые)	-	-	-	-
JO метка	Переход, если O=1	-	-	-	-
JNO метка	Переход, если O=0	-	-	-	-
LOOP метка	CX := CX-1, переход, если CX<>0	-	-	-	-
CALL метка	Вызов подпрограммы	-	-	-	-
RET	Возврат из подпрограммы	-	-	-	-
PUSH регистр	Запись регистра в стек	-	-	-	-
POP регистр	Восстановление из стека	-	-	-	-
PUSHF	Запись регистра FLAGS в стек	-	-	-	-
POPF	Восстановление FLAGS из стека	-	-	-	-
CLC	Сброс C	-	-	0	-
STC	Установка C	-	-	1	-
CBW	Заполняет AX знаковым битом AL				
CWD	Заполняет DX знаковым битом AX				