Министерство образования Российской Федерации
НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Лисицин Д.В.

Программирование на языке ассемблера. Части 1, 2

НОВОСИБИРСК
2018

Рецензенты:
канд. техн. наук, доцент И.Л. Еланцева
канд. техн. наук, доцент Ю.В. Тракимус

Работа подготовлена на кафедре
теоретической и прикладной информатики
для студентов 3-го курса ФПМИ

Учебное пособие представляет собой первую часть курса программирования на языке ассемблера, в нем рассматриваются основы языка ассемблера, а также интерфейс с языком C++.

Учебное пособие предназначено для студентов, обучающихся по направлениям «Прикладная математика и информатика» и «Математическое обеспечение и администрирование информационных систем».

Введение

Большинство программистов пишут свои программы на языках высокого уровня, таких как C++. Подобные языки специально разработаны для того, чтобы приблизить конструкции, с помощью которых программист общается с ЭВМ, к предметной области, для которой предназначена конкретная программа.

В отличие от этих языков, язык ассемблера является машинно-ориентированным языком, это символический аналог машинного языка, позволяющий программировать на уровне машинных команд. Если при разработке языков высокого уровня стремились избежать ориентации на специальные технические особенности вычислительной техники, то язык ассемблера предназначен как раз для учета специфики конкретного компьютера.

Язык ассемблера имеет два достоинства: с одной стороны, он позволяет писать программы на уровне команд микропроцессора, а с другой стороны, не требует запоминания числовых кодов команд, которыми оперирует процессор. Язык ассемблера оперирует символическими мнемоническими кодами вместо машинных команд и описательными именами для полей данных и адресов памяти.

Языки высокого уровня позволяют программисту абстрагироваться от особенностей компьютера. В этом смысле они похожи на автомобили с автоматической коробкой передач. Однако человек действует более гибко, более искусно, чем автоматическая коробка передач.

Язык ассемблера представляет собой для ЭВМ эквивалент ручной коробки передач. Он обеспечивает программисту больший контроль над ЭВМ за счет большего труда, большей детализации и меньшего удобства.

Прикладные программы, полностью написанные на языке ассемблера, встречаются довольно редко, поскольку на их создание, отладку и последующее сопровождение уходит слишком много времени. Поэтому язык ассемблера в основном используется при написании отдельных фрагментов прикладных программ, т.е. там, где требуется максимальная скорость их работы или непосредственный доступ к оборудованию.

Программы на языке ассемблера часто пишут для встраиваемых компьютерных систем. В качестве примеров можно привести системы питания и зажигания автомобилей, системы управления кондиционерами, охранные системы, системы управления полетами, электронные записные книжки, модемы, принтеры и другие «умные» устройства, содержащие встроенный микропроцессор.

Язык ассемблера часто используется в программах для систем реального времени, для игровых приставок, а также для разработки системного программного обеспечения, в том числе для разработки драйверов устройств (низкоуровневых системных программ, напрямую взаимодействующих с обслуживаемыми ими устройствами).

Однако, и «высокоуровневым» программистам полезно знание языка ассемблера, оно поможет лучше понять методики взаимодействия между аппаратным обеспечением компьютера, операционной системой и прикладными программами.

Подобно C++ язык ассемблера представляет собой набор слов, задающих ЭВМ действия, которые она должна выполнить. Но в отличие от языка C++ слова из набора команд языка ассемблера имеют непосредственное отношение к компонентам ЭВМ. Так как ассемблер требует задания действий на уровне компонент ЭВМ, то программисту необходимо понимать свойства и возможности интегральной микросхемы, содержащей эти компоненты, а именно микропроцессора ЭВМ.

Важным отличием языка ассемблера от языков высокого уровня является то, что написанные на нем программы не являются переносимыми. Учитывая изложенные выше моменты, язык ассемблера не может быть переносимым по определению, поскольку он тесно связан с архитектурой микропроцессоров определенного семейства. Таким образом, на сегодняшний день существует много совершенно разных языков ассемблера. Каждый из них привязан либо к конкретному семейству процессоров, либо к конкретной архитектуре компьютера.

В данном курсе мы будем в основном рассматривать программирование на языке ассемблера для 32-разрядных процессоров фирмы Intel и 32-разрядной операционной системы Microsoft Windows. Предполагается, что работа с программами ведется в Microsoft Visual C++.

1. Регистры микропроцессора

Регистрами называют участки высокоскоростной памяти, расположенные внутри процессора и предназначенные для оперативного хранения данных и быстрого доступа к ним со стороны внутренних компонентов процессора. Рассмотрим основные регистры.

1.1. Регистры общего назначения

Имеется восемь 32-разрядных регистров общего назначения EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP. Они предназначены для хранения данных и адресов, программист может их использовать для реализации своих алгоритмов. Возможно обращение к младшим 16 битам регистров как к регистрам с именами AX, BX, CX, DX, SI, DI, BP, SP соответственно. В свою очередь регистры AX, BX, CX, DX позволяют отдельно обращаться к своим старшим и младшим восьми байтам как к регистрам AH/AL, BH/BL, CH/CL и DH/DL.

Структура регистра EAX представлена на рисунке 1 (структура регистров EBX, ECX, EDX полностью идентична, а в регистрах ESI, EDI, EBP, ESP нельзя обращаться по-отдельности к их 8-битным частям).

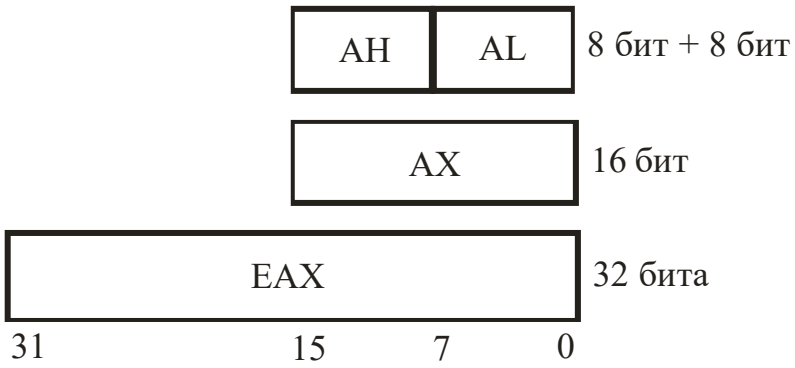

Рисунок 1.

В принципе, все эти регистры доступны для хранения операндов без особых ограничений, хотя в определенных условиях некоторые из них имеют жесткое функциональное назначение, закрепленное на уровне логики работы машинных команд.

1.2. Сегментные регистры

Эти регистры используются в качестве базовых при обращении к заранее распределенным областям оперативной памяти, которые называются сегментами. Существует три типа сегментов и, соответственно, сегментных регистров:

– кода (CS), в них хранятся только команды процессора, т.е. машинный код программы;

– данных (DS, ES, FS и GS), в них хранятся области памяти, выделяемые под данные;

– стека (SS).

Стек представляет собой область памяти, используемую для временного хранения данных и адресов. Микропроцессор использует стек для хранения адреса возврата из текущей подпрограммы, но стек можно использовать также для восстановления содержимого регистров, изменяемых при работе программы.

Работа с памятью особо проста, если используется плоская (несегментированная) модель памяти FLAT. Каждой программе может быть выделен один большой сегмент размером до 4 Гбайт (обычно программисту доступно около половины), а сегменты, описываемые в программе, являются логическими. Адреса ячеек памяти являются 32-битными (смещение внутри сегмента), а сегментные регистры программист не использует. Мы будем использовать именно модель памяти FLAT.

Среди описанных выше регистров общего назначения особо следует выделить регистр ESP. В нем хранится указатель на вершину стека, поэтому его не рекомендуется использовать для хранения каких-либо операндов программы.

1.3. Регистр командного указателя

Смещение на команду, которая должна быть выполнена следующей, содержит 32-битный регистр EIP (instruction pointer), его младшие 16 битов составляют регистр IP. Этот регистр непосредственно недоступен программисту, но загрузка и изменение его значения производятся различными командами управления, к которым относятся команды условных и безусловных переходов, вызова процедур и возврата из процедур.

1.4. Регистр флагов

Разрядность регистра флагов EFLAGS равна 32 битам. Отдельные биты данного регистра имеют определенное функциональное назначение и называются флагами. К младшим 16 битам регистра EFLAGS можно обратиться как к регистру FLAGS.

Флаги определяют текущее состояние машины и результаты выполнения команд. Многие арифметические команды и команды сравнения изменяют состояние флагов. Рассмотрим назначение наиболее важных для программиста флагов (все они находятся в регистре FLAGS):

OF (overflow flag – флаг переполнения) – указывает на переполнение при командах для чисел со знаком (0 – нет переполнения, 1 – переполнение произошло);

SF (sign flag – флаг знака) – содержит результирующий знак после операций над числами со знаком (0 – плюс, 1 – минус);

ZF (zero flag – флаг нуля) – показывает, является ли результат нулевым (0 – ненулевой, 1 – нулевой результат);

PF (parity flag – флаг четности) – показывает четность младших 8 бит данных (1 – четное и 0 – нечетное число);

CF (carry flag – флаг переноса) – указывает на перенос из старшего бита результата после арифметических операций (0 – нет переноса, 1 – перенос был).

2. Структура программы на языке ассемблера

Программа на языке ассемблера представляет собой последовательность операторов: команд или директив (псевдооператоров), описывающих выполняемые действия. Команды языка ассемблера представляют собой краткую нотацию системы команд микропроцессора. Директивы сообщают программе, производящей трансляцию, что ей делать с вводимыми командами и данными. Кстати, эту программу называют ассемблером, что объясняет происхождение термина – «язык ассемблера»

Команды и директивы могут включать в себя операции, которые дают ассемблеру информацию об операндах, относительно которых нет полной определенности. Существует несколько видов операций, например, арифметические, логические.

2.1. Команды

Команда языка ассемблера в исходной программе имеет следующий вид (в квадратных скобках здесь и ниже указываются необязательные элементы):

[Метка:] Мнемокод [Операнд(ы)] [; Комментарий]

Метка, команда и операнд разделяются, по крайней мере, одним пробелом или символом табуляции. Приведем пример:

```
M:    MOV        AX, BX      ; Метка, команда, два операнда
```

Метка в языке ассемблера может состоять из одного или нескольких символов и содержать латинские буквы, цифры и некоторые специальные символы: _, ?, $, @. Первым символом в метке должна быть буква или специальный символ.

Поле *мнемокода* содержит имя (мнемонический код) команды микропроцессора. Мнемокод указывает ассемблеру, какое действие должен выполнить микропроцессор. Например, MOV – имя команды пересылки данных.

Если команда специфицирует выполняемое действие, то *операнд* определяет начальное значение данных или элементы, над которыми выполняется действие по команде. Например, в приведенной выше команде MOV указано, что надо скопировать содержимое регистра BX в регистр AX.

Команда может иметь один или несколько операндов, или вообще быть без операндов. Если операнды присутствуют, то они отделяются от мнемокода, по крайней мере, одним пробелом или символом табуляции, между собой операнды разделяются запятой.

Обычно в командах с двумя операндами первый из них представляет собой приемник, а второй – источник. Операнд-источник определяет значение, которое берется микропроцессором для сложения, вычитания, сравнения со значением операнда-приемника или просто для загрузки в операнд-приемник. Поэтому при исполнении команды операнд-источник никогда не изменяется, в то время как операнд-приемник изменяется почти всегда

В нашем случае у команды MOV операнд-приемник – регистр AX, а операнд источник – регистр BX.

Комментарий начинается на любой строке исходного модуля с символа «точка с запятой» (;). Ассемблер полагает в этом случае, что все символы, находящиеся справа от «;», являются комментарием. Комментарий может занимать всю строку или следовать за командой на той же строке.

2.2. Директивы

Ассемблер имеет ряд директив (или псевдооператоров), которые позволяют управлять процессом трансляции. Они действуют только в процессе трансляции программы и не генерируют машинных кодов.

Рассмотрим некоторые важные директивы.

2.2.1. Директива COMMENT

Символом «точка с запятой» (;) можно задать только однострочный комментарий. При необходимости многострочного комментария необходимо в начале каждой строки указывать этот символ.

Другим решением здесь будет блочный комментарий, начинающийся с директивы COMMENT, за которой следует символ комментария, определяемый программистом. При этом компилятор игнорирует все строки, расположенные между директивой COMMENT и символом, указанным программистом. Например:

```
COMMENT !
Это строка комментария.
А вот еще одна строка комментария.
!
```

2.2.2. Директивы SEGMENT и ASSUME

Любые ассемблерные программы содержат, по крайней мере, один сегмент – сегмент кода.

Директива SEGMENT служит для описания сегмента и имеет следующий формат:

```
имя SEGMENT [параметры]
.    .    .    .    .
имя ENDS
```

Имя сегмента должно обязательно присутствовать и быть уникальным. Директива ENDS означает конец сегмента. Обе директивы SEGMENT и ENDS должны иметь одинаковые имена. Директива SEGMENT может содержать параметры, которые используются в программах, имеющих сложную структуру, в том числе многомодульных. Мы их рассматривать не будем.

При использовании директивы SEGMENT требуется указать компилятору, с каким сегментом связан тот или иной сегментный регистр.

Директива ASSUME сообщает ассемблеру назначение каждого сегмента. Он имеет формат

```
ASSUME SS: имя_сегмента, DS: имя_сегмента, CS: имя_сегмента,
ES: имя_сегмента, FS: имя_сегмента, GS: имя_сегмента
```

Если программа не использует какой-либо регистр, то его описание можно опустить.

Для модели памяти FLAT следует использовать следующее описание

```
ASSUME    SS: FLAT, DS: FLAT, CS: FLAT, ES: FLAT, FS: ERROR,
GS: ERROR
```

Здесь «FLAT» – это имя группы, в которую объединяются сегменты стека, данных и кода.

2.2.3. Упрощенные директивы сегментации

Для простых программ можно упростить описание их сегментации. Для этого используются упрощенные директивы сегментации. Сегменты, описанные с их использованием, имеют предопределенные имена.

Совместно с упрощенными директивами сегментации используется директива указания модели памяти MODEL, которая частично управляет размещением сегментов и выполняет функции директивы ASSUME (поэтому при введении в программу упрощенных директив сегментации директиву ASSUME можно не указывать). Формат:

.MODEL модель памяти [, язык]

Директива указания модели памяти, как и упрощенные директивы сегментации, начинается с точки.

Обязательным параметром директивы MODEL является модель памяти. Этот параметр определяет модель сегментации памяти для программного модуля. Мы почти всегда будем рассматривать модель памяти FLAT.

Параметр «язык» определяет порядок передачи параметров при вызове процедур, а также соглашение о присвоении имен процедурам и внешним идентификаторам. Необходимость в использовании этого параметра обычно появляется при написании программ на разных языках программирования.

При использовании упрощенных директив сегментации предполагается, что программный модуль может иметь только определенные типы сегментов. Используются следующие директивы описания сегментов (указываются после директивы .MODEL):

.CODE [имя] – начало или продолжение сегмента кода (необязательный параметр замещает имя _TEXT, заданное по умолчанию);

.DATA – начало или продолжение сегмента инициализированных данных;

.DATA? – начало или продолжение сегмента неинициализированных данных (главным преимуществом директивы является то, что при ее использовании уменьшается размер исполняемого файла);

.CONST – начало или продолжение сегмента данных, в котором определены константы (сегмент имеет атрибут «только для чтения»);

.STACK [размер] – начало или продолжение сегмента стека с (необязательно) указанным размером памяти, который должен быть выделен под область стека.

Упомянем еще об одной обязательной директиве – директиве задания набора команд. Она определяет тип процессора, для которого создается программа (программа будет работать и на более старших моделях). Возможные значения .386 (процессор 80386), .486 (процессор 80486), .586 (процессор Pentium), .686 (процессор Pentium Pro).

2.2.4. Директива END

Если директива ENDS завершает сегмент, то директива END полностью завершает всю программу:

END [метка точки входа]

Операнд может быть опущен, если программа не содержит точку входа. Например, если эта программа должна быть скомпонована с другим (главным) модулем. Для обычной программы с одним модулем операнд содержит имя метки, с которой начинается выполнение программы.

Пример. Описание структуры простой программы с использованием упрощенных директив сегментации.

```
.386   ; в программе используются команды процессора 80386
.MODEL     FLAT
.DATA
; Описание данных
.CODE
START:
; Код программы
END START        ; Метка точки входа – START
```

2.2.5. Директива PROC

Сегмент кода может содержать одну или несколько процедур, которые определяются директивой PROC.

```
имя_процедуры   PROC
.       .       .       .       .
                RET
имя_процедуры   ENDP
```

Директива ENDP определяет конец процедуры и имеет имя, аналогичное имени в директиве PROC.

Команда RET завершает выполнение процедуры.

Когда микропроцессор вызывает процедуру, то он помещает в стек адрес возврата (содержимое регистра EIP). Команда RET заставит микропроцессор извлечь из стека двойное слово и поместить его в регистр EIP.

Пример. Структура программы, состоящей из одной процедуры.

```
.386
.MODEL     FLAT
.DATA
; Описание данных
.CODE
BEGIN       PROC
; Код процедуры
BEGIN       ENDP
END         BEGIN
```

2.2.6. Директивы определения данных

Для хранения переменных в программах используют ячейки памяти.

Перечислим основные типы целочисленных данных:
BYTE – байт, 8-разрядное беззнаковое целое;
SBYTE – байт, 8-разрядное знаковое целое;
WORD – слово, 16-разрядное беззнаковое целое;
SWORD – слово, 16-разрядное знаковое целое;
DWORD – двойное слово, 32-разрядное беззнаковое целое;
SDWORD – двойное слово, 32-разрядное знаковое целое;
QWORD – учетверенное слово, 64-разрядное целое.

В памяти числа представляются в двоичной системе счисления. Биты нумеруются, начиная с 0, причем 0-й бит является младшим. Числа со знаком хранятся в дополнительном коде, причем старший бит (7-й в байте, 15-й в слове и т.д.) содержит знак. Ноль в знаковом бите соответствует положительному числу, а единица – отрицательному (см. рисунок 2).

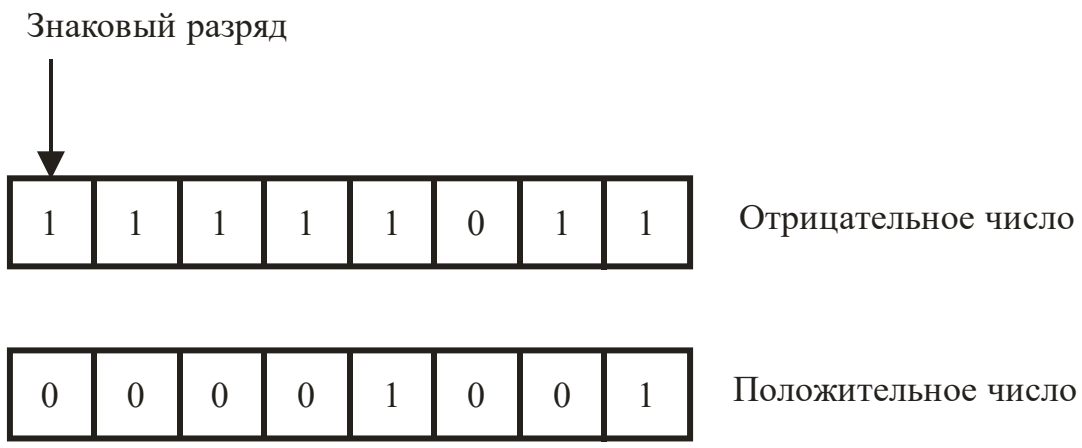

Рисунок 2.

В таблицах 1, 2 приведены диапазоны возможных значений для беззнаковых и знаковых типов соответственно.

Таблица 1

Тип	Диапазон значений
Байт	0…255
Слово	0…65535
Двойное слово	0…4294967295
Учетверенное слово	0…18446744073709551615

Таблица 2

Тип	Диапазон значений

Байт	–128…+127
Слово	–32768…+32767
Двойное слово	–2147483648…+2147483647
Учетверенное слово	–9223372036854775808… +9223372036854775807

Память для переменных указанных типов резервируют следующие директивы:

DB (define byte – определить байт);

DW (define word – определить слово);

DD (define double word – определить двойное слово);

DQ (define quadword – определить учетверенное слово).

Также могут использоваться директивы, одноименные типам, их основное отличие – возможность описания переменных знакового типа.

Директивы имеют следующий формат:

[имя] директива выражение [,...]

Операнд *выражение* может принимать одну из нескольких форм.

Выражение может быть константой.

```
VD   DB   127D    ; Десятичный формат
VD1  DB   127     ; Десятичный формат по умолчанию
VH   DB   0FFFH   ; Шестнадцатеричный формат
VB   DB   1100B   ; Двоичный формат
VO   DB   253O    ; Восьмеричный формат
VO1  DB   253Q    ; Восьмеричный формат
```

Суффикс в записи константы определяет систему счисления. Если основание в целочисленной константе не указано, предполагается, что число десятичное. Если шестнадцатеричная константа начинается с буквы, перед ней должен ставиться символ нуля, чтобы ассемблер не воспринял эту константу как идентификатор.

Заметим, что с многоразрядными двоичными числами очень трудно работать, поскольку их тяжело анализировать. Поэтому при представлении двоичных чисел в ассемблерной программе и отладчике обычно используется шестнадцатеричная форма записи. Каждая цифра в шестнадцатеричном числе представляет собой 4 бита (тетраду), а 2 шестнадцатеричные цифры вместе составляют 1 байт.

Пример. Двоичное число 000101101010011110010100 можно представить в шестнадцатеричной форме как 16A794, соответствие шестнадцатеричных цифр и тетрад приведено в таблице 3.

Таблица 3

1	6	A	7	9	4
0001	0110	1010	0111	1001	0100

Часто для большей наглядности записи двоичных чисел тетрады отделяют друг от друга пробелом:

0001 0110 1010 0111 1001 0100

Тогда перевести число из двоичной формы в шестнадцатеричную не составит особого труда.

Отрицательные числа записываются со знаком минус:

```
VD2  DB   –32
```

Но они могут быть также прямо записаны в дополнительном коде.

Имя переменной, указываемое перед названием директивы, является меткой, значением которой является адрес данной переменной (смещение относительно начала сегмента).

Директивы можно использовать для создания в памяти таблиц (массивов), при этом элементы таблицы перечисляются через запятую. Следующие последовательности задают две таблицы по 4 элемента в каждой, одна из байтов, а другая из слов:

```
B_TABLE      DB      0,0,0,8          ; Таблица байтов
W_TABLE      DW      0,1000,500,0     ; Таблица слов
```

Обратите внимание на то, что первые три элемента таблицы B_TABLE имеют одинаковые значения. У ассемблера есть операция DUP (duplicate – дублировать), которая позволяет повторять операнды, не набирая их каждый раз заново. С помощью операции DUP определение таблицы B_TABLE можно записать следующим образом:

```
B_TABLE      DB      3 DUP(0),8       ; Таблица байтов
```

При определении неинициализированной переменной в поле *выражение* указывается вопросительный знак (?). Например, следующий оператор резервирует байт, но не заносит в него какое-либо значение:

```
TEMP         DB      ?
```

Можно зарезервировать ячейки памяти для целой таблицы. Например, оператор

```
TABLE        DB      12 DUP(?)        ; Таблица байтов
```

зарезервирует 12 байт памяти.

Директива DB может воспринимать в качестве выражения также *строку символов.* Это позволяет заносить в память сообщения об ошибках, заголовки таблиц и всякий другой текст. Для этого текст надо заключить в одинарные или двойные кавычки. Приведем пример:

```
MSG          DB      'Ошибка'
```

Каждый символ строки занимает один байт памяти, причем ассемблер размещает в памяти последовательность ASCII-кодов, соответствующих каждому символу строковой константы.

Если внутри строковой константы должен использоваться символ одинарной или двойной кавычки, это делается так, как показано ниже:

```
"Буква 'a' -- первый символ алфавита"
'Он воскликнул: "Привет!", и зашел в комнату.'
```

Переменные используются также для хранения адресов ячеек памяти, на которые могут ссылаться команды программы.

Поскольку адреса являются 32-битными, для их хранения надо использовать директиву DD. Например, оператор

```
L_N             DD          L1
```

выделит память под 32-битовый адрес метки L1 и присвоит ячейке памяти имя L_N. Содержащую адрес переменную называют *указателем.*

В процессорах фирмы Intel при хранении данных в памяти используется так называемый прямой порядок следования байтов. Это означает, что младший байт переменной хранится в памяти по меньшему адресу. Оставшиеся байты переменной хранятся в последующих ячейках памяти в порядке возрастания их старшинства.

В качестве примера рассмотрим двойное слово, значение которого равно 12345678H. Предположим, что оно хранится в памяти со смещением 0. Тогда значение 78H будет храниться в первом байте со смещением 0, 56H – во втором байте со смещением 1, 34H – в третьем байте со смещением 2, 12H – в четвертом байте со смещением 3 (см. рисунок 3).

Значение	78	56	34	12
Смещение	0000	0001	0002	0003

Рисунок 3.

2.2.7. Директива ALIGN

Директива ALIGN используется для выравнивания адреса переменной на границу байта, слова, двойного слова, учетверенного слова или параграфа (т.е. 16-ти байтов). Ее синтаксис следующий:

```
ALIGN граница
```

Здесь вместо параметра *граница* следует подставить число 1, 2, 4, 8 или 16. Если значение параметра равно 1, что адрес следующей за этой директивой переменной выравнивается на границу байта (т.е. не выравнивается вовсе). Это значение принято по умолчанию. Если значение параметра равно 2, то следующая за директивой ALIGN переменная выравнивается на границу слова (т.е. располагается с четного адреса). Если значение параметра равно 4, то следующая переменная выравнивается на границу двойного слова (т.е. ее адрес делится на 4) и т.д. При необходимости ассемблер автоматически пропускает после директивы ALIGN необходимое количество байтов, чтобы расположить переменную по нужному адресу.

Зачем вообще нужно выравнивать данные? Дело в том, что процессор может обрабатывать данные гораздо быстрее, если они выровнены соответствующим образом. Например, если адрес двойного слова кратен 4, т.е. выровнен на

границу двойного слова, доступ к нему осуществляется за 1 машинный цикл, а если нет, то за 2.

Пример.

```
BVAL    BYTE    ?
ALIGN   2
WVAL    WORD    ?
BVAL2   BYTE    ?
ALIGN   4
DVAL    DWORD   ?
DVAL2   DWORD   ?
```

Можно уменьшить использование директивы ALIGN (или вовсе его избежать), если описывать переменные в порядке уменьшения их размера.

```
ALIGN   4
DVAL    DWORD   ?
DVAL2   DWORD   ?
WVAL    WORD    ?
 BVAL   BYTE    ?
BVAL2   BYTE    ?
```

2.3. Особенности разработки 16-разрядных программ под MS-DOS

Для 16-разрядных приложений код и данные хранятся в отдельных (физических) сегментах размером до 64 Кбайт.

Начальные адреса этих сегментов содержатся в четырех регистрах сегментов: регистре сегмента команд (CS), регистре сегмента стека (SS), регистре сегмента данных (DS), регистре дополнительного сегмента (ES).

Микропроцессор использует 20-битовые адреса ячеек оперативной памяти. Адрес каждой ячейки задается двумя числами: номером блока (адресом начала сегмента из регистра сегмента) и смещением. Физический адрес образуется путем добавления 16-битового смещения к номеру блока, умноженному на 16. Таким образом,

$$\text{физический адрес} = \text{смещение} + 16 \times (\text{регистр сегмента}).$$

Имея в своем распоряжении 20-битовый адрес, микропроцессор может адресовать 1 Мбайт.

Смещение вершины стека хранит регистр SP. Адрес вершины стека определяется парой регистров SS:SP.

Смещение команды, которая должна быть выполнена следующей, содержит регистр IP. Адрес следующей исполняемой команды определяется парой регистров CS:IP.

Основной моделью памяти, для которой разрабатываются программы на ассемблере, является малая (SMALL), согласно которой имеется всего два сегмента – сегмент кода и объединенный сегмент данных и стека. Рассмотрением этой модели мы и ограничимся.

Пример. Программа, состоящая из одной процедуры.

```
.MODEL  SMALL
```

```
.DATA
        SOURCE  DB   10
        DEST    DB   ?
.CODE
BEGIN   PROC    FAR
        MOV     AX, @DATA
        MOV     DS, AX
        MOV     AL, SOURCE          ; Скопировать значение
        MOV     DEST, AL            ; SOURCE в DEST
        MOV     AX, 4C00H           ; Выход из программы
        INT     21H
BEGIN   ENDP
END     BEGIN
```

Комментарии к программе.

1. Система DOS инициализирует все сегментные регистры, кроме DS. Поэтому в начало программы (после метки точки входа) нужно вставить две дополнительные команды, предназначенные для занесения в регистр DS адреса сегмента данных программы. Для этого используется встроенный идентификатор @DATA.

2. Процедуре всегда приписан один из двух атрибутов дистанции. NEAR (близкая процедура) и FAR (далекая процедура). Он указывается в качестве операнда оператора PROC. Если операнд опущен, то подразумевается атрибут NEAR.

Процедура с атрибутом NEAR может быть вызвана только из того сегмента команд, в котором она определена.

Программу можно вызвать из операционной системы DOS или из отладчика. Но так как и DOS, и отладчик размещены в сегментах команд, отличных от сегмента команд программы, то ее основная процедура должна иметь атрибут FAR.

3. В системе DOS имеется набор функций, предоставляющих пользователю множество удобных возможностей взаимодействия с клавиатурой, монитором, принтером, диском и др. Работа с этими функциями основана на механизме прерываний, сходном с вызовом процедур. Вызов прерывания осуществляется командой INT (interrupt – прерывать) с операндом – номером прерывания. Для вызова функций DOS используется прерывание 21H, номер функции при этом загружается в регистр AH.

Для выхода из программы мы воспользовались функцией 4CH, в регистр AL при этом поместили нулевой код завершения.

Назовем еще две функции прерывания 21H: A – чтения строки с клавиатуры, 9 – вывода строки на экран.

3. Команды пересылки данных

3.1. Команда MOV

Команда MOV (move – переслать) пересылает байт, слово или двойное слово между регистром и ячейкой памяти или между двумя регистрами. Она может также пересылать непосредственно адресуемое значение (константу) в регистр или в ячейку памяти.

Напомним, что команда MOV имеет следующий формат:

MOV приемник, источник

В ней допустимо большинство из возможных сочетаний операндов.

Нельзя осуществить непосредственную пересылку данных из одной ячейки памяти в другую. Чтобы выполнить такую пересылку, данные источника надо загрузить в регистр общего назначения, а затем запомнить содержимое этого регистра в приемнике.

```
MOV     AL, SOURCE      ; Скопировать значение
MOV     DEST, AL        ; SOURCE в DEST
```

3.2. Команда XCHG

Команда XCHG (exchange data – обмен данными) позволяет обменять содержимое двух операндов. Формат команды:

XCHG приемник, источник

Для операндов команды XCHG нужно соблюдать те же правила и ограничения, что и для операндов команды MOV, за исключением того, что операнды команды XCHG не могут быть непосредственно заданными значениями.

Примеры:

```
XCHG    AX, BX      ; Обмен содержимого 16-разрядных регистров
XCHG    AH, AL      ; Обмен содержимого 8-разрядных регистров
XCHG    VAL, BX     ; Обмен содержимого 16-разрядного операнда
                    ; в памяти и регистра BX
XCHG    EAX, EBX    ; Обмен содержимого 32-разрядных регистров
```

Чтобы поменять содержимое двух переменных, расположенных в памяти, необходимо воспользоваться промежуточным регистром и двумя дополнительными командами MOV:

```
.DATA
        VAL1    DW  1
        VAL2    DW  2
.       .   .   .   .
. CODE
        MOV     EAX, VAL1
        XCHG    EAX, VAL2
        MOV     VAL1, EAX
```

3.3. Команды работы со стеком

Перечислим основные причины использования стека в программах.

– Стек представляет собой очень удобное место для сохранения значения регистров, в случае если они используются для нескольких целей. После измене-

ния содержимого регистра, его первоначальное значение можно легко восстановить.

– При вызове процедуры процессор сохраняет в стеке адрес следующей за ней команды. Тем самым обеспечивается возврат из процедуры.

– При вызове процедуры ей обычно передается ряд входных параметров, которые могут быть помещены в стек.

– Сразу после вызова внутри процедуры может быть создан ряд локальных переменных, которые удобно расположить в стеке. Значение этих переменных теряется при возврате управления в вызвавшую процедуру.

Команда PUSH помещает содержимое регистра, ячейки памяти или константу размером в слово или двойное слово на вершину стека. А команда POP снимает слово или двойное слово с вершины стека и помещает его в ячейку памяти или регистр.

Команды PUSH и POP имеют следующие форматы:

PUSH источник

POP приемник

Под вершиной стека мы понимаем ячейку памяти, адрес которой содержится в указателе стека ESP. Стек «растет» по направлению к младшим адресам памяти (к ячейке 0), поэтому первый помещаемый в стек элемент запоминается в ячейке стека с наибольшим адресом, следующий – на два или четыре байта младше и т.д.

Регистр ESP всегда указывает на элемент, помещенный в стек последним. Следовательно, команда PUSH вычитает 4 или 2 из значения указателя стека, а затем пересылает операнд-источник в стек. Действуя обратным образом, команда POP пересылает в операнд-приемник элемент, адрес которого содержится в регистре ESP, а затем добавляет 4 или 2 к содержимому этого регистра.

Воздействие команд PUSH и POP на стек отразим на рисунках 4 и 5.

1. Команда PUSH.

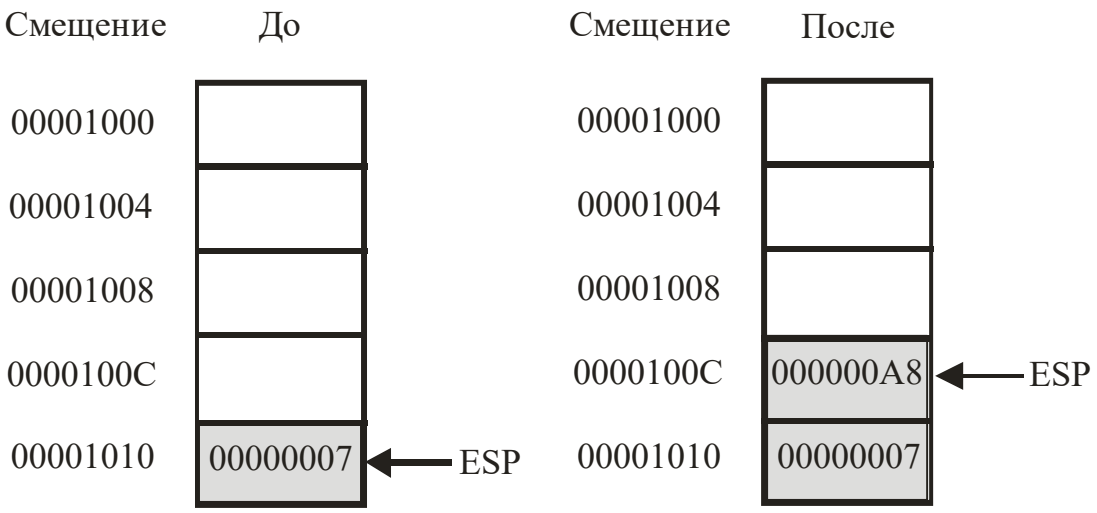

Рисунок 4.

2. Команда POP.

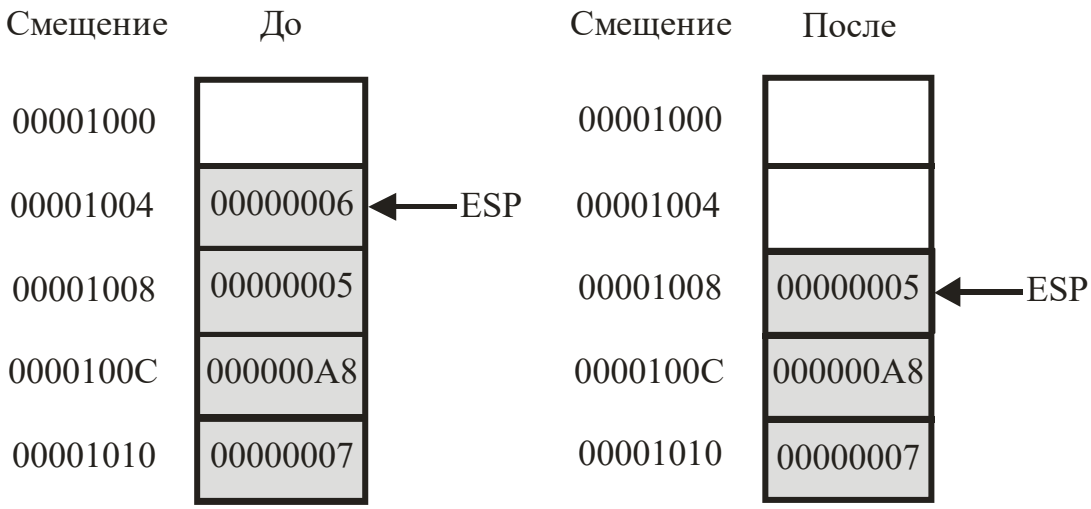

Рисунок 5.

Команды PUSHFD и POPFD пересылают содержимое регистра флагов EFLAGS в стек и обратно. Команды PUSHF и POPF пересылают содержимое регистра флагов FLAGS в стек и обратно. Они идентичны командам PUSH и POP, но в них не требуется указывать операнд, так как под ним подразумевается регистр флагов. Как и в случае команд PUSH и POP, данные команды всегда используются парами.

Команда PUSHAD пересылают содержимое регистров общего назначения в стек, команда POPAD выполняет обратную операцию. В них также не требуется указывать операнд. Команды PUSHA и POPA делают то же самое для регистров AX, BX, CX, DX, SI, DI, BP, SP. В случае с регистрами ESP и SP в PUSHAD и PUSHA используется значение, которое находилось в регистре до начала работы команды. А в командах POPAD и POPA помещенное в стек значение ESP или SP игнорируется.

Команды PUSHA, PUSHAD обычно используется в начале процедуры или фрагмента кода, в котором модифицируется много регистров общего назначения. Для восстановления первоначального значения этих регистров в конце процедуры или фрагмента кода используется команда POPA или POPAD.

Пример.

```
MYSUB     PROC
          PUSHAD    ; Сохраним в стеке регистры общего назначения
.         .    .    .    .
          MOV       EAX, . . .
          MOV       EDX, . . .
          MOV       ECX, . . .
.         .    .    .    .
          POPAD     ; Восстановим значения регистров
          RET
MYSUB     ENDP
```

3.4. Команда обмена байтов

Команда обмена байтов BSWAP имеет формат

BSWAP регистр32

Она изменяет порядок следования байтов в указанном 32-разрядном регистре: из порядка «младший–старший» в порядок «старший–младший» (переставляет нулевой и третий, первый и второй байты).

4. Арифметические команды

Микропроцессор может выполнять арифметические команды над двоичными числами со знаком или без знака. Имеются команды для выполнения четырех стандартных действий арифметики – сложения, вычитания, умножения и деления.

При выполнении процессором арифметических команд может возникнуть ошибка переполнения, если значения операндов слишком малы или слишком велики. В языках высокого уровня ситуация целочисленного переполнения обычно полностью игнорируется, что иногда приводит к трудно выявляемым ошибкам в процессе выполнения программы. В отличие от этого, в языке ассемблера у вас под рукой находятся все средства для отслеживания и обработки подобных ситуаций, поскольку вы всегда сможете проконтролировать состояние флагов процессора после выполнения каждой арифметической команды.

4.1. Команды сложения

Команды ADD (add – сложить) и ADC (add with carry – сложить с переносом) могут складывать 8-, 16- и 32-битовые операнды. Формат команд:

ADD приемник, источник

Команда ADD складывает содержимое операнда-источника и операнда-приемника и помещает результат в операнд-приемник. В символической нотации ее действия можно описать как

приемник = приемник + источник

Пример. Команда

```
ADD         AX, CX
```

складывает 16-битовые значения регистров AX и CX и возвращает результат в регистр AX.

Команда ADC делает то же, что и команда ADD, но при сложении использует также флаг переноса CF, что можно записать следующим образом:

приемник = приемник + источник + перенос

Флаг CF имеет большое значение при выполнении процессором арифметических операций с беззнаковыми целыми числами. Данный флаг устанавливается в случае, если результат выполнения такой операции очень велик (или очень мал)

и поэтому он не помещается в выделенное для него пространство операнда – приемник данных.

Пример. 8-битовый регистр может содержать значения без знака в диапазоне от 0 до 255. Если мы выполним двоичное сложение чисел 250 и 10, то получим 260 (см. рисунок 6).

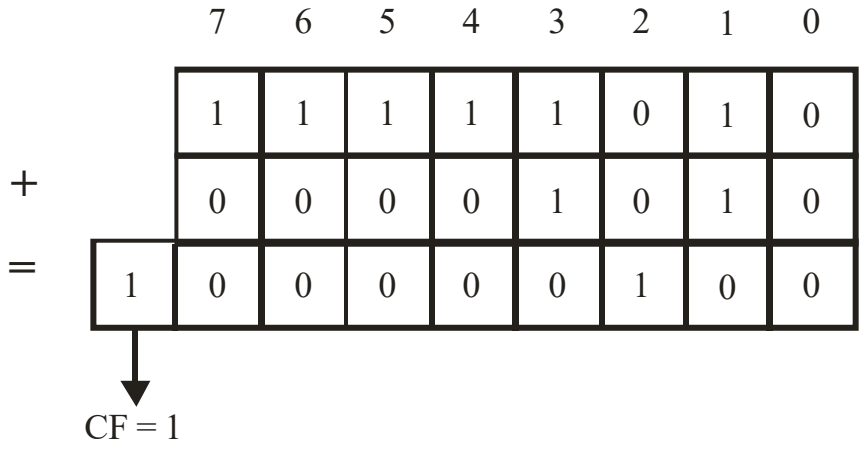

Рисунок 6.

Как видим, результат занимает 9 двоичных битов. Если при выполнении этой операции мы использовали 8-битовые регистры, то младшие 8 битов будут занесены в регистр-приемник, а девятый бит – во флаг переноса CF.

Складывать можно также значения *повышенной точности*, каждое из которых располагается в нескольких регистрах или ячейках памяти. Команда ADD используется тогда для сложения младших частей значений повышенной точности, а команда ADC – для сложения их старших частей.

Пример. Если операнды имеют длину более 32 битов, то можно воспользоваться следующим приемом. Пусть складываются 64-битовое число, находящееся в регистрах ECX и EDX, с 64-битовым числом, находящимся в регистрах EAX и EBX.

```
ADD  EAX, ECX  ; Сначала сложить младшие 32 бита,
ADC  EBX, EDX  ; а затем старшие 32 бита
```

Использованная здесь команда ADC добавляет к (EDX)+(EBX) любой перенос от сложения (ECX)+(EAX).

Можно добавлять находящийся в памяти операнд к регистру и наоборот или добавлять непосредственный операнд к регистру или операнду, находящемуся в памяти. Однако нельзя добавить значение одной ячейки памяти к другой или использовать в качестве приемника непосредственное значение.

Команды ADD и ADC могут воздействовать на ряд флагов.

Флаг переноса CF равен 1, если результат сложения не помещается в операнде-приемнике; в противном случае он равен 0.

Флаг четности PF равен 1, если 8 младших битов результата имеют четное число битов со значением 1; в противном случае он равен 0.

Флаг нуля ZF равен 1, если результат равен 0; в противном случае он равен 0.

Флаг знака SF равен 1, если результат отрицателен (старший бит равен 1); в противном случае он равен 0.

Флаг переполнения OF учитывается только при выполнении арифметических операций с целыми числами со знаком.

Флаг переполнения OF равен 1, если сложение двух чисел одного знака (оба положительные или оба отрицательные) приводит к результату, который превышает диапазон допустимых значений приемника в обратном коде, а сам приемник при этом меняет знак. В противном случае флаг OF равен 0.

Процессор определяет, возникло ли в результате выполнения арифметической операции целочисленное переполнение чисто механически. Для этого он сравнивает значение двух битов переноса, которые получились в результате выполнения операции: флага переноса CF и бита переноса в знаковый разряд. Если значения этих битов не равны, устанавливается флаг переполнения.

Например, при сложении двух двоичных чисел 10000000B и 11111110B не возникает переноса из 6-го в 7-й (знаковый) разряд, но при этом возникает перенос из знакового разряда во флаг CF (см. рисунок 7). Поскольку значения этих флагов не равны, устанавливается флаг OF.

	7	6	5	4	3	2	1	0
CF = 1 ←	1	0	0	0	0	0	0	0
+	1	1	1	1	1	1	1	0
=	0	1	1	1	1	1	1	0

Рисунок 7.

4.2. Команда XADD

Команда XADD имеет формат

XADD приемник, источник

Команда сначала производит обмен данных (выполняет команду XCHG), а затем выполняет команду ADD.

4.3. Команда приращения значения приемника на единицу

Команда INC (increment – прирастить) добавляет 1 к содержимому регистра или ячейки памяти, но не воздействует на флаг переноса CF (в отличие от команды ADD).

4.4. Команды вычитания

Команды SUB (substract – вычесть) и SBB (substract with borrow – вычесть с заемом) аналогичны соответственно командам сложения ADD и ADC. Однако при вычитании флаг переноса CF действует как признак *заема.* Формат команд:

SUB приемник, источник

Команда SUB вычитает операнд-источник из операнда-приемника и возвращает результат в операнд-приемник, т.е.

приемник = приемник – источник

Команда SBB делает то же самое, но дополнительно вычитает значение флага переноса CF:

приемник = приемник – источник – перенос

Как и в случае сложения, команды вычитания выполняют две отдельные функции. Первая команда SUB вычитает числа размером в байт, слово или двойное слово, а также младшие биты чисел повышенной точности. Другая команда SBB вычитает старшие биты чисел повышенной точности. Например, команда

```
SUB  AX, CX
```

вычитает содержимое регистра CX из содержимого регистра AX и возвращает результат в регистр AX.

Если размеры операндов превышают 32 бита, то пользуйтесь последовательностью команд вида

```
SUB  EAX, ECX  ; Вычесть младшие 32 бита
SBB  EBX, EDX  ; а затем – старшие 32 бита
```

Здесь мы вычитаем 64-битовое число, помещенное в регистры ECX и EDX, из 64-битового числа, помещенного в регистры EAX и EBX. При вычитании содержимого регистра EDX из содержимого регистра EBX команда SBB учитывает возможность заема при выполнении первого вычитания.

Можно вычитать из содержимого регистра содержимое ячейки памяти (и наоборот) или вычитать из содержимого регистра либо ячейки памяти непосредственное значение. Нельзя непосредственно вычесть значение одной ячейки из другой или использовать непосредственное значение как приемник.

При выполнении команды вычитания процессор заменяет ее на команду сложения, инвертируя при этом значение исходного операнда. Например, вместо операции 4 – 1 выполняется операция 4 + (–1). Напомним, что для представления отрицательных чисел в процессорах Intel используется двоичный дополнительный код. Поэтому –1 представляется в виде двоичного числа 11…11B.

Команды SUB и SBB могут воздействовать на ряд флагов.

Флаг переноса CF равен 1, если требуется заем; в противном случае он равен 0.

Флаг четности PF равен 1, если результат вычитания имеет четное число младших восьми битов со значением 1; в противном случае он равен 0.

Флаг нуля ZF равен 1, если результат равен 0; в противном случае он равен 0.

Флаг знака SF равен 1, если результат отрицателен (старший бит равен 1); в противном случае он равен 0.

Флаг переполнения OF равен 1, если при вычитании чисел, имеющих разные знаки, результат превышает диапазон значений приемника в обратном коде, а сам приемник изменяет знак; в противном случае флаг OF равен 0.

4.5. Команда уменьшения содержимого приемника на единицу

Команда DEC (decrement – уменьшить) вычитает 1 из содержимого регистра или ячейки памяти, но при этом не воздействует на флаг переноса CF (в отличие от команды SUB).

4.6. Команда обращения знака

Команда NEG вычитает значение операнда-приемника из нулевого значения. Команда NEG оказывает на флаги то же действие, что и команда SUB.

Команда NEG полезна для вычитания значения регистра или ячейки памяти из непосредственного значения.

Пример. Необходимо вычесть значение регистра AL из 100. Так как непосредственное значение не может служить приемником, то команда

```
SUB  100, AL
```

недопустима. В качестве альтернативы можно обратить знак содержимого регистра AL и добавить к нему 100:

```
NEG  AL
ADD  AL, 100
```

4.7. Команды умножения

Команда MUL (multiply – умножить) умножает числа без знака, а IMUL (integer multiply – умножить целые числа) – числа со знаком. Обе команды могут умножать байты, слова и двойные слова.

Эти команды имеют формат

```
MUL источник
IMUL источник
```

где *источник* – регистр общего назначения или ячейка памяти размером в байт, слово, или двойное слово.

В качестве второго операнда команды MUL и IMUL используют содержимое регистра AL (при операциях над байтами), регистра AX (при операциях над словами) или регистра EAX (при операциях над двойными словами).

Произведение имеет *двойной размер* и возвращается следующим образом:

Умножение *байтов* возвращает 16-битовое произведение в регистрах AH (старший байт) и AL (младший байт).

Умножение *слов* возвращает 32-битовое произведение в регистрах DX (старшее слово) и AX (младшее слово).

Умножение *двойных слов* возвращает 64-битовое произведение в регистрах EDX (старшее слово) и EAX (младшее слово).

По завершении исполнения этих команд флаги переноса CF и переполнения OF показывают, какая часть произведения существенна для последующих операций.

После исполнения команды MUL флаги CF и OF равны 0, если старшая половина произведения равна 0; в противном случае оба этих флага равны 1.

После исполнения команды IMUL флаги CF и OF равны 0, если старшая половина произведения представляет собой лишь расширение знака младшей половины. В противном случае они равны 1.

Команды MUL и IMUL не позволяют в качестве операнда использовать непосредственное значение. Такое значение перед умножением надо загрузить в регистр или в ячейку памяти.

Пример. В результате исполнения команд

```
MOV     DX, 10
MUL     DX
```

содержимое регистра AX будет умножено на 10.

Команда IMUL имеет также следующие расширенные форматы.

В команде IMUL с двумя операндами первый операнд является первым сомножителем, в него же записывается результат, второй операнд является вторым сомножителем. При этом первый операнд может быть 16- или 32-разрядным регистром, а второй операнд может быть регистром, ячейкой памяти или непосредственным значением того же размера (непосредственное значение может быть также 8-разрядным).

В команде IMUL с тремя операндами первый операнд является результатом умножения, второй и третий операнды являются сомножителями. При этом первый операнд может быть 16- или 32-разрядным регистром, а второй операнд может быть регистром или ячейкой памяти того же размера, третий операнд может быть только непосредственным значением того же размера или 8-разрядным.

Если результат выполнения этих команд слишком большой, то он усекается, а флаги OF и CF становятся равными 1, в противном случае флаги OF и CF становятся равными 0.

4.8. Команды деления

Имея две отдельные команды умножения, микропроцессор имеет и две отдельные команды деления.

Команда DIV (divide – разделить) выполняет деление чисел без знака, а команда IDIV (integer divide – разделить целые числа) выполняет деление чисел со знаком. Эти команды имеют формат

DIV источник

IDIV источник

где *источник* – делитель размером в байт, слово или двойное слово. Он находится в регистре общего назначения или в ячейке памяти. Делимое должно иметь *двойной размер*. Оно извлекается из регистров AH и AL (при делении на 8-битовое число), из регистров DX и AX (при делении на 16-битовое число) или из регистров EDX и EAX (при делении на 32-битовое число).

Результаты возвращаются следующим образом:

Если операнд-источник представляет собой байт, то частное возвращается в регистре AL, а остаток в регистре AH.

Если операнд-источник представляет собой слово, то частное возвращается в регистре AX, а остаток – в регистре DX.

Если операнд-источник представляет собой двойное слово, то частное возвращается в регистре EAX, а остаток – в регистре EDX.

Обе команды оставляют состояние флагов неопределенными, но если частное не помещается в регистре-приемнике (AL, AX или EAX), то микропроцессор генерирует прерывание «деление на 0».

Команды DIV и IDIV не позволяют прямо разделить на непосредственное значение; его надо предварительно загрузить в регистр или ячейку памяти.

Пример. Команды

```
MOV BX, 20
DIV  BX
```

разделят объединенное содержимое регистров DX и AX на 20.

4.9. Команды преобразования типа

Что делать, если размеры операндов, участвующих в арифметических операциях, разные? Например, предположим, что в операции сложения один операнд занимает слово, а другой – двойное слово. Однако, в операции сложения должны участвовать операнды одного формата. Если числа беззнаковые, то можно на базе исходного операнда сформировать новый операнд (формата двойного слова), старшие разряды которого просто заполнить нулями. Для чисел со знаком в системе команд процессора есть так называемые команды преобразования типа. Эти команды расширяют байты в слова, слова – в двойные слова и двойные слова – в учетверенные слова, заполняя старшие биты вновь формируемого операнда значениями знакового бита исходного объекта. Эта операция приводит к целым значениям того же знака и той же величины, что и исходная, но уже в более длинном формате.

Команда CBW (convert byte to word – преобразовать байт в слово) воспроизводит 7-й бит регистра AL во всех битах регистра AH (см. рисунок 8).

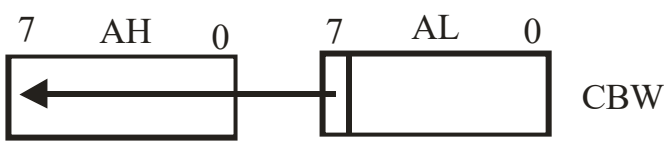

Рисунок 8.

Команда CWD (convert word to double-word – преобразовать слово в двойное слово) воспроизводит 15-й бит регистра AX во всех битах регистра DX (см. рисунок 9).

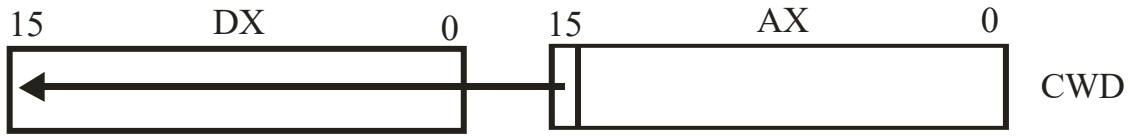

Рисунок 9.

Таким образом, команда CBW позволяет сложить байт и слово, вычесть слово из байта и т.д. Аналогично команда CWD позволяет разделить слово на слово. Приведем несколько примеров:

```
CBW             ; Сложить байт в AL со словом в BX
ADD     AX, BX
CBW             ; Умножить байт в AL на слово в BX
IMUL    BX
CWD             ; Разделить слово в AX на слово в BX
IDIV    BX
```

Команда CWDE (преобразовать слово в двойное слово) воспроизводит 15-й бит регистра AX в старших 16 битах регистра EAX.

Команда CDQ (преобразовать двойное слово в учетверенное слово) воспроизводит 31-й бит регистра EAX во всех битах регистра EDX.

Имеются также две команды пересылки, попутно производящие преобразование типа.

Команда пересылки с распространением знака MOVSX имеет формат:

MOVSX приемник, источник

Команда расширяет 8- или 16-разрядное значение источника, которое может быть регистром или операндом в памяти, до 16- или 32-разрядного значения в приемнике – одном из регистров, используя знаковый бит для заполнения старших позиций значения приемника (см. рисунок 10). Данную команду удобно использовать для подготовки операнда *со знаком* к выполнению арифметических операций.

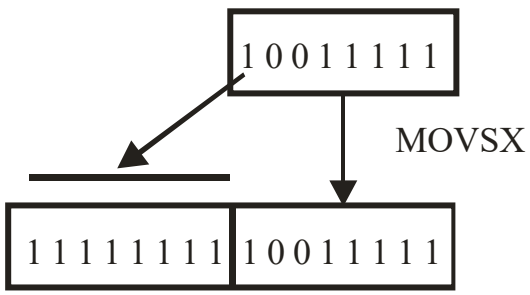

Рисунок 10.

Команда пересылки с расширением нулем MOVZX имеет формат:

MOVZX приемник, источник

и операнды тех же типов, что у предыдущей команды.

Она расширяет 8- или 16-разрядное значение источника до 16- или 32-разрядного с заполнением нулями старших позиций значения приемника (см. рисунок 11). Данную команду удобно использовать для подготовки операнда *без знака* к выполнению арифметических действий.

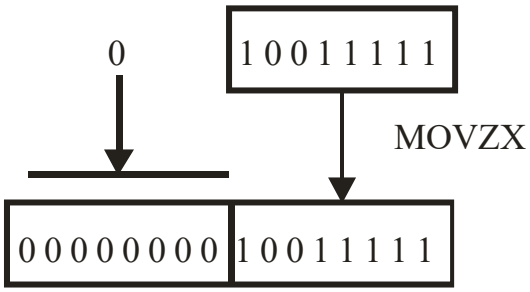

Рисунок 11.

Пример. Вычислим значение (A + B)/C, где A, B, C – байтовые знаковые переменные.

```
MOV     AL, A
CBW
MOVSX   BX, B
ADD     AX, BX
IDIV    C           ; в AL – частное, в AH – остаток
```

5. Режимы адресации

5.1. Регистровая и непосредственная адресация

При *регистровой* адресации микропроцессор извлекает операнд из регистра или загружает его в регистр.

Например, команда

```
MOV     AX, CX
```

копирует 16-битовое содержимое регистра CX в регистр AX. В данном примере микропроцессор использует регистровую адресацию для извлечения операнда-источника из регистра CX и загрузки его в регистр-приемник AX.

Непосредственная адресация позволяет указывать 8-, 16- или 32-битовое значение константы в качестве операнда-источника. Эта константа содержится в команде, а не в регистре или в ячейке памяти.

Например, команда

```
MOV         CX, 5
```

загружает значение 5 в регистр CX.

5.2. Эффективный адрес

Смещение, которое вычисляется для доступа к находящемуся в памяти операнду, называется *эффективным адресом* операнда. Эффективный адрес показывает, на каком расстоянии (в байтах) располагается операнд от начала сегмента.

Эффективный адрес может быть задан несколькими способами. В общем случае мы можем определить эффективный адрес операнда как сумму, состоящую из нескольких слагаемых:

– базы, представляющей собой содержимое одного из регистров общего назначения;

– индекса, представляющего собой содержимое одного из регистров общего назначения, кроме ESP; индекс может быть умножен на масштабный множитель, равный 1, 2, 4 или 8;

– смещения, представляющего собой 8-, 16- или 32-разрядное значение.

Схематически это можно представить так, как показано на рисунке 12. Формула вычисления эффективного адреса (EA) выглядит как

EA = База + (Индекс × Множитель) + Смещение

База		Индекс		Множитель		Смещение
EAX EBX ECX EDX ESP EBP ESI EDI	+	EAX EBX ECX EDX EBP ESI EDI	×	1 2 4 8	+	8 бит 16 бит 32 бита

Рисунок 12.

Следует заметить, что база, индекс и смещение могут применяться в любых комбинациях, причем любой компонент может отсутствовать. Масштабирующий множитель применяется только с индексом.

Рассмотрим некоторые характерные комбинации компонентов для получения эффективного адреса.

5.3. Прямая адресация

При прямой адресации эффективный адрес является составной частью команды (смещением).

Обычно прямая адресация применяется, если операндом служит метка.

Пример.

```
VAR     DB   10
.       .    .    .    .
MOV     AL, VAR
```

Команда MOV загружает содержимое ячейки памяти VAR в регистр AL. Сама метка VAR имеет значение адреса этой ячейки.

При задании операнда команды к имени метки можно добавлять дополнительное смещение. Такая конструкция используется в программе для доступа к ячейкам памяти, которым не была назначена метка.

Рассмотрим массив байтов, которому присвоена метка ARRAY:

```
ARRAY   DB   10H, 20H, 30H, 40H, 50H
```

Если указать метку ARRAY в качестве источника данных в команде MOV, то в результате будет выбрано значение первого байта этого массива:

```
MOV     AL, ARRAY       ; AL = 10H
```

Чтобы обратиться ко второму байту массива, нужно к смещению, соответствующему метке ARRAY, прибавить единицу:

```
MOV     AL, ARRAY + 1   ; AL = 20H
```

Таким же образом можно обращаться к отдельным байтам, словам, двойным словам и ячейкам памяти, имеющим больший размер.

```
W_VAR   DW   ?
.       .    .    .    .
MOV     AL, W_VAR + 1   ; обращаемся к старшему байту слова W_VAR
```

5.4. Косвенная регистровая адресация

При косвенной регистровой адресации эффективный адрес операнда содержится в регистре общего назначения. Косвенные регистровые операнды надо заключать в квадратные скобки, чтобы отличать их от регистровых операндов.

Например, команда

```
MOV     AL, [EBX]
```

загружает в регистр AL содержимое ячейки памяти, адресуемой значением регистра EBX.

Как поместить адрес в регистр EBX? Для этого применяются операция OFFSET (смещение) и команда LEA (load effective address – загрузить эффективный адрес).

Пусть в сегменте данных описана строка S1:

```
S1      DB   "Assembly"
```

Операция OFFSET возвращает адрес переменной или метки. Например, оператор

```
MOV     EBX, OFFSET S1
```

загрузит адрес переменной S1 в регистр EBX.

Здесь в регистр EBX помещается адрес строки, который одновременно является и адресом первого элемента (имеющего индекс 0).

Таким образом, для загрузки первого элемента строки S1 в регистр AL можно воспользоваться последовательностью команд

```
MOV     EBX, OFFSET S1
MOV     AL, [EBX]
```

Команда LEA пересылает адрес ячейки памяти в 32-битовый регистр. Она имеет формат

LEA регистр, память

Таким образом, предыдущие две команды можно заменить на следующие:

```
LEA     EBX, S1
MOV     AL, [EBX]
```

Алгоритм выполнения этого фрагмента программного кода показан на рисунке 13.

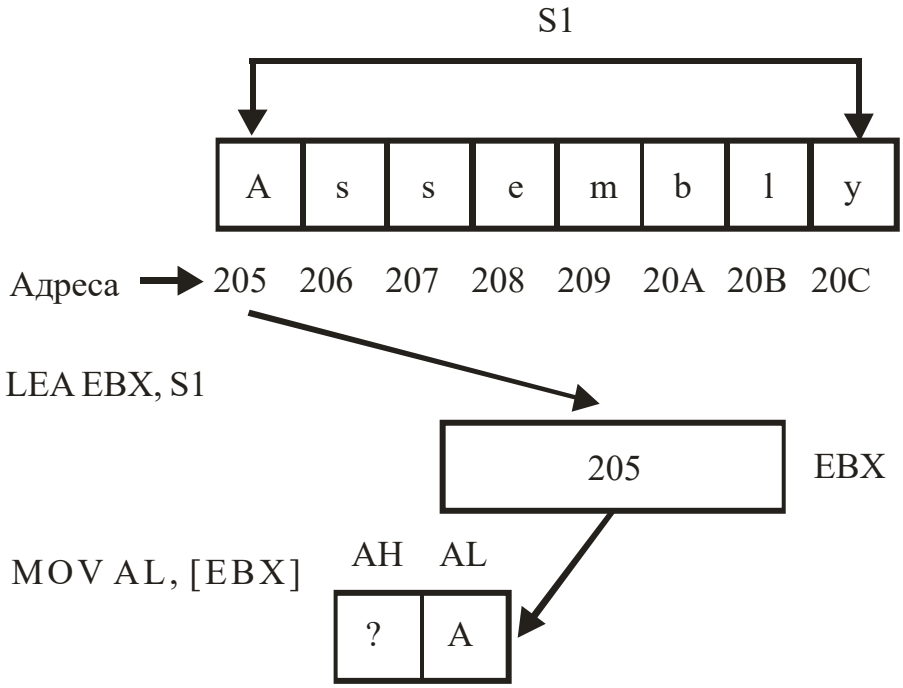

Рисунок 13.

Заметим, что операнд операции OFFSET должен быть известен на этапе трансляции, в то время как соответствующий операнд команды LEA должен быть известен только к моменту выполнения команды микропроцессором. В частности, командой LEA можно определить текущее смещение косвенного операнда.

Однако те же действия, что и в предыдущих двух фрагментах, выполняет одна команда

```
MOV     AL, S1
```

с той разницей, что в предыдущих случаях содержимое регистра EBX уничтожается.

Если нужен доступ лишь к одной ячейке памяти (в данном случае S1), то разумнее воспользоваться одной командой. Однако для доступа к *нескольким*

ячейкам, начиная с данного базового адреса, гораздо лучше иметь эффективный адрес в регистре.

5.5. Адресация по базе

При адресации по базе ассемблер вычисляет эффективный адрес с помощью сложения значения смещения с содержимым регистра общего назначения (базового регистра).

Адресацию по базе удобно использовать при доступе к структурированным записям данных, которые расположены в разных областях памяти. В этом случае адрес начала записи помещается в базовый регистр и доступ к ее отдельным элементам осуществляется по их смещению относительно базы. А для доступа к разным записям, имеющим одну и ту же структуру, достаточно соответствующим образом изменять содержимое базового регистра.

Пример. Предположим, что требуется прочитать учетные записи для ряда работников. При этом каждая запись содержит табельный номер работника, номер отдела, номер группы, возраст, тарифную ставку и т.д.

Если номер отдела хранится в пятом и шестом байтах записи, а начальный адрес записи содержится в регистре EBX, то команда

```
MOV     AX, [EBX]+4
```

загрузит в регистр AX номер отдела, в котором служит данный работник (сдвиг равен 4, а не 5, потому что первый байт записи имеет номер 0).

Пример.

```
S1      DB    "Assembly"
.       .     .       .       .
LEA     EBX, S1
MOV     AL, [EBX]+4
```

Здесь команда LEA помещает в регистр EBX адрес строки. Во время выполнения второй команды к содержимому регистра EBX прибавляется значение 4, указывая на пятый элемент строки S1 (это символ m), после чего значение этого символа помещается в регистр AL. Таким образом, после выполнения второй команды регистр AL будет содержать символ m. Алгоритм выполнения этого фрагмента программного кода показан на рисунке 14.

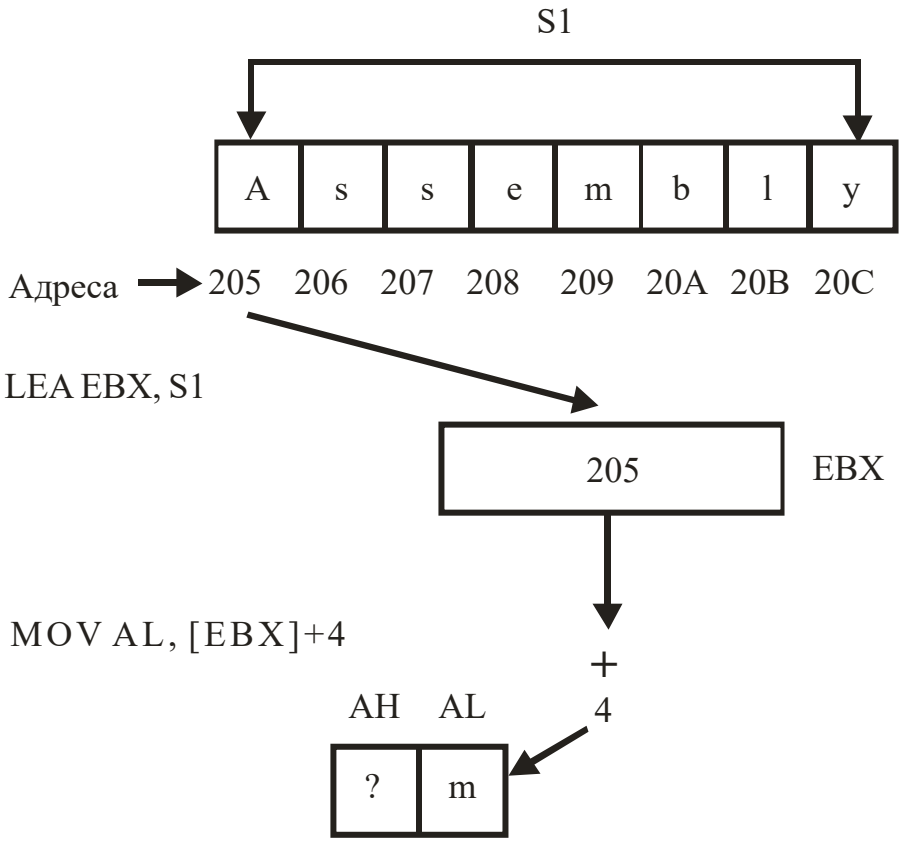

Рисунок 14.

Ассемблер позволяет указывать адресуемые по базе операнды разными способами. Следующие команды эквивалентны предыдущей:

```
MOV     AX, [EBX]+4
MOV     AX, [EBX+4]
MOV     AX, [EBX][4]
```

5.6. Прямая адресация с индексированием

При прямой адресации с индексированием эффективный адрес вычисляется как сумма значений смещения и индекса, в качестве которого может выступать любой регистр общего назначения, кроме ESP. Индекс может быть отмасштабирован с использованием множителя. Этот тип адресации удобен для доступа к элементам таблицы, когда смещение указывает на начало таблицы, а индекс – на ее элемент.

Пример. Последовательность команд

```
S1      DB    "Assembly"
.       .     .       .       .
MOV     EDI, 4
MOV     AL, S1[EDI]
```

загрузит пятый элемент строки S1 (символ m) в регистр AL.

В таблице слов соседние элементы отстоят друг от друга на два байта, поэтому при работе с ней надо *удваивать номер элемента* при вычислении значения

индекса. Аналогично в таблице двойных слов номер элемента надо умножать на 4, а в таблице учетверенных слов – на 8.

Если TABLE – таблица слов, то для загрузки в регистр AX ее третьего элемента надо использовать последовательность команд

```
MOV     EDI, 4
MOV     AX, TABLE [EDI]
```

Более простым в этом случае является использование множителя при индексе. Множитель может быть равен 1, 2, 4, 8 и используется следующим образом

```
MOV     EDI, 2
MOV     AX, TABLE [EDI*2]
```

5.7. Адресация по базе с индексированием

При адресации по базе с индексированием эффективный адрес вычисляется как сумма значений базового регистра, (отмасштабированного или нет) индекса и, возможно, смещения.

Так как в этом режиме адресации складывается два отдельных смещения, то он удобен при адресации двумерных массивов.

Заметим, что специальных средств для описания такого типа данных в ассемблере нет. Двухмерный массив нужно моделировать. На описании самих данных это почти никак не отражается – память под массив выделяется с помощью директив резервирования и инициализации памяти.

Непосредственно обработка двумерных массивов производится в сегменте кода, где программист, описывая алгоритм обработки на ассемблере, определяет, что некоторую область памяти необходимо трактовать как двухмерный массив. При этом мы вольны в выборе того, как понимать расположение элементов двухмерного массива в памяти: по строкам или по столбцам.

Чтобы отразить при описании двумерную структуру массивов можно следовать следующим примерам.

```
MASS    DB 3 DUP(5 DUP(?))
TABLE   DB 10H,  20H,  30H,  40H,  50H
        DB 60H,  70H,  80H,  90H,  0A0H
        DB 0B0H, 0C0H, 0D0H, 0E0H, 0F0H
```

В памяти эти массивы располагаются в виде непрерывной последовательности байтов, так как если бы они были одномерными массивами. Однако мы считаем, что логически каждая их этих последовательностей байтов организована в виде двумерного массива, содержащего три логические строки и пять логических столбцов.

Если последовательность однотипных элементов в памяти трактуется как двухмерный массив, расположенный по строкам, то адрес элемента (i, j) вычисляется по формуле

адрес_массива + (количество_столбцов × i + j) × размер_элемента.

При этом возможны два основных варианта выбора компонентов для формирования эффективного адреса:

– в качестве смещения указывается адрес массива, в базовый регистр загружается смещение строки относительно начала массива, а в индексный регистр – смещение элемента в текущей строке

```
MOV     AL, TABLE[EBX][ESI]
```

– в базовый регистр загружается адрес строки, а в индексный регистр – смещение элемента в текущей строке

```
MOV     AL, [EBX][ESI]
```

Пример. Во втором из описанных массивов на пересечении второй строки и третьего столбца находится число 80H. В приведенных ниже фрагментах вычисляется его адрес. Предположим, что наш двумерный массив располагается по адресу 250. Тогда текущий адрес нашего элемента массива будет равен 257.

В первом варианте (см. рисунок 15) в регистр EBX загружается смещение второй строки относительно начала таблицы – произведение количества столбцов на номер строки (т.е. 5), а в регистр ESI – номер столбца (т.е. 2):

```
MOV     EBX, 5                  ; Смещение строки
MOV     ESI, 2                  ; Номер столбца
MOV     AL, TABLE[EBX + ESI]    ; 250 + 5 + 2 = 257
```

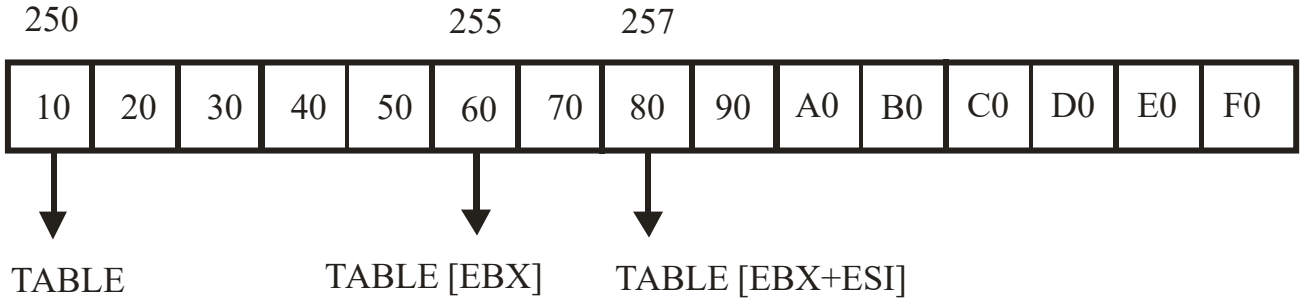

Рисунок 15.

Следующие команды эквивалентны предыдущей:

```
MOV     AL, [TABLE + EBX + ESI]
MOV     AL, [EBX + TABLE][ESI]
MOV     AL, [EBX][ESI + TABLE]
MOV     AL, TABLE[EBX][ESI]
```

Во втором варианте (см. рисунок 16) в регистр EBX загружается адрес начала таблицы, затем к нему прибавляется смещение второй строки относительно начала таблицы, в результате определяется адрес начала нужной нам строки, а в регистр ESI загружается номер столбца:

```
MOV     EBX, OFFSET TABLE
ADD     EBX, 5      ; Количество_столбцов*номер_строки
MOV     ESI, 2      ; Номер столбца
```

```
MOV         AL, [EBX + ESI]
```

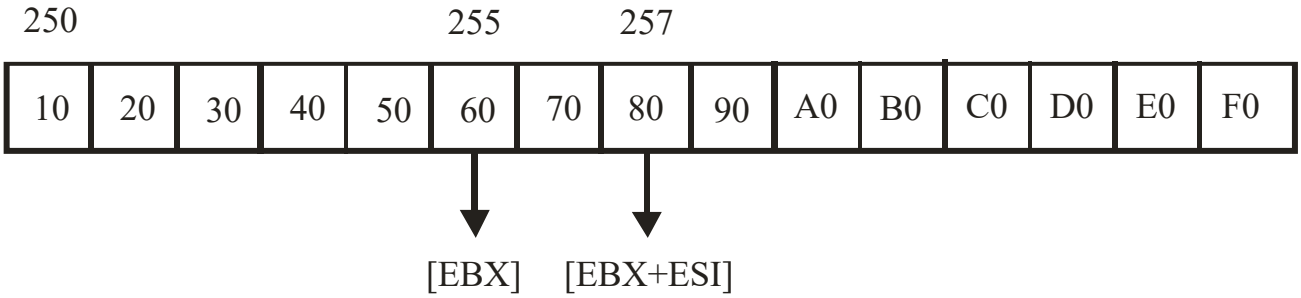

Рисунок 16.

6. Команды передачи управления и сравнения

6.1. Команды CALL и RET

Команды, обеспечивающие исполнение процедур, должны выполнять три функции:

1. Обеспечить сохранение содержимого регистра EIP.
2. Заставить микропроцессор начать исполнение процедуры.
3. Использовать сохраненное содержимое регистра EIP для возврата в программу и обеспечить продолжение ее исполнения с этого места.

Все эти функции выполняются двумя командами: CALL (вызвать процедуру) и RET (возвратиться из процедуры).

Команда CALL осуществляет функции запоминания адреса возврата и передачи управления процедуре. Она помещает в стек адрес возврата (содержимое регистра EIP) и загружает в EIP адрес начала процедуры.

Команда CALL имеет формат

CALL имя

где *имя* – имя вызываемой процедуры, т.е. метка ее начала.

Команда RET заставляет микропроцессор возвратиться из процедуры в программу, вызвавшую эту процедуру. Делается это «откатом» всего, что сделала команда CALL: команда RET извлекает из стека адрес возврата и загружает его в указатель команд EIP. Команда RET обязательно должна быть последней командой процедуры, исполняемой микропроцессором.

Команда RET имеет формат

RET [число]

где необязательный параметр *число* определяет число байтов, которые извлекаются из стека между удалением адреса возврата и передачей управления на адрес возврата. Таким способом удобно очищать стек от аргументов процедуры.

Можно осуществлять также *косвенный* вызов процедуры через регистр или ячейку памяти.

Можно вызвать процедуру *через регистр*, например, следующим образом:

```
CALL        EBX
```

В данном случае регистр EBX содержит адрес процедуры. При исполнении этой команды микропроцессор копирует содержимое регистра EBX в указатель команд EIP, затем передает управление команде, адресуемой этим регистром.

При косвенных вызовах *через ячейку памяти* микропроцессор извлекает значение указателя команд EIP для процедуры из сегмента данных.

Процедуру можно вызвать косвенно, используя переменную размером в двойное слово, например:

```
CALL    MEM_DWORD                   ; прямая адресация
CALL    DWORD PTR [EBX]             ; косвенная адресация
CALL    DWORD PTR [EBX] [ESI]       ; адресация по базе
                                    ; с индексированием
CALL    DWORD PTR PROCEDURES [EBX] ; прямая адресация
                                        ; с индексированием
```

Операция указателя PTR позволяет изменить у операнда *атрибут типа* (BYTE, WORD, DWORD и др.). В данном случае эта операция используется для доступа к двойным словам.

Процедура может сама вызывать другие процедуры. Таким образом, возможно использование вложенных процедур. Возможны и рекурсивные процедуры. Так как каждая команда CALL помещает в стек четыре байта адреса, то число уровней вложения ограничено только размером стека.

6.2. Команда безусловного перехода JMP

Команда JMP (jump unconditionally – перейти безусловно) представляет собой эквивалент на языке ассемблера оператора GOTO; она заставляет микропроцессор извлечь новую команду не из следующей ячейки памяти, а из какой-то другой.

Команда JMP имеет формат

```
JMP имя
```

где операнд *имя* подчиняется тем же правилам, что и операнд команды CALL. Иначе говоря, он может быть прямым или косвенным. При выполнении этой команды микропроцессор загружает в регистр EIP адрес команды, на которую необходимо перейти.

Например, команда

```
JMP   M1
```

осуществляет прямой переход на метку M1.

6.3. Команды условной передачи управления

У микропроцессора есть группа команд, которые позволяют ему «принять решение» о ходе исполнения программы в зависимости от определенных условий. Например, в зависимости от нулевого значения регистра или единичного значения какого-либо флага. Если такое условие выполнено, то микропроцессор вы-

полнит переход. В противном случае он продолжит исполнение со следующей команды программы.

Команды условной передачи управления имеют следующий общий формат:

Jx метка

где x – модификатор, состоящий из одной или нескольких букв (jump – перейти).

Для знаковых и беззнаковых данных предназначены разные команды.

В таблице 4 приведены значения аббревиатур в названии команды.

Таблица 4

Мнемоническое обозначение	Оригинальный термин	Перевод	Тип операндов
E	Equal	Равно	Любые
N	Not	Нет	Любые
G	Greater	Больше	Числа со знаком
L	Less	Меньше	Числа со знаком
A	Above	Выше (в смысле больше)	Числа без знака
B	Below	Ниже (в смысле меньше)	Числа без знака

Команды перехода для беззнаковых и знаковых данных приведены в таблицах 5 и 6 соответственно.

Таблица 5

Команды	Описание	Значения флагов
JE/JZ	Переход, если равно/нуль	ZF = 1
JNE/JNZ	Переход, если не равно/не нуль	ZF = 0
JA/JNBE	Переход, если выше/не ниже или равно	ZF = 0 и CF = 0
JAE/JNB	Переход, если выше или равно/не ниже	CF = 0
JB/JNAE	Переход, если ниже/не выше или равно	CF = 1
JBE/JNA	Переход, если ниже или равно/не выше	CF = 1 или ZF = 1

Таблица 6

Команды	Описание	Значения флагов
JE/JZ	тоже	тоже
JNE/JNZ	тоже	тоже
JG/JNLE	Переход, если больше/не меньше или равно	ZF = 0 и SF = OF
JGE/JNL	Переход, если больше или равно/не меньше	SF = OF
JL/JNGE	Переход, если меньше/не больше или равно	SF ≠ OF
JLE/JNG	Переход, если меньше или равно/не больше	ZF = 1 или SF ≠ OF

Любую проверку можно кодировать одним из двух мнемонических кодов. Например, JB и JNAE генерирует один и тот же объектный код.

В таблице 7 приведены команды перехода, использующие специальные арифметические проверки.

Таблица 7

Команды	Описание	Значения флагов
JS	Переход, если есть знак (отрицательно)	SF = 1
JNS	Переход, если нет знака (положительно)	SF = 0
JC	Переход, если есть перенос	CF = 1
JNC	Переход, если нет переноса	CF = 0
JO	Переход, если есть переполнение	OF = 1
JNO	Переход, если нет переполнения	OF = 0
JP	Переход, если сумма битов четная	PF = 1
JNP	Переход, если сумма битов не четная	PF = 0
JCXZ	Переход, если в регистре CX нуль	CX = 0
JECXZ	Переход, если в регистре ECX нуль	ECX = 0

Отметим одно ограничение, свойственное командам JCXZ и JECXZ: они могут адресовать только *короткие* переходы в пределах от –128 байт до +127 байт от *следующей* за ними команды.

6.4. Команда CMP

Командам условной передачи управления могут предшествовать любые команды, изменяющие состояния флагов, но чаще они используются совместно с командами сравнения.

Основная команда сравнения CMP (compare – сравнить) имеет формат

CMP приемник, источник

Команда CMP вычитает операнд-источник из операнда-приемника и в зависимости от результата устанавливает или обнуляет флаги. Значения флагов для операндов без знака указаны в таблице 8, для операндов со знаком – в таблице 9.

Таблица 8

Условие	Флаги	
	ZF	CF
приемник больше источника	0	0
приемник равен источнику	1	0
приемник меньше источника	0	1

Таблица 9

Условие	Флаги	
	ZF	SF, OF
приемник больше источника	0	SF = OF
приемник равен источнику	1	
приемник меньше источника	0	SF ≠ OF

Какими условными переходами надо пользоваться при возможных сочетаниях значений источника и приемника указано в таблице 10.

Таблица 10

Условие перехода	Следующая за CMP команда	
	для чисел без знака	для чисел со знаком
приемник больше источника	JA	JG
приемник равен источнику	JE	JE
приемник не равен источнику	JNE	JNE
приемник меньше источника	JB	JL
приемник меньше источника или равен ему	JBE	JLE
приемник больше источника или равен ему	JAE	JGE

Пример. В следующем фрагменте кода выполняется переход на метку NEXT при равенстве нулю содержимого регистра EAX. Равенство нулю содержимого EAX определяется при помощи команды CMP, которая воздействует на флаги.

```
        CMP  EAX, 0
        JZ      NEXT
        ; Обработка ситуации, когда EAX не равен 0
NEXT:
        ; Обработка ситуации, когда EAX равен 0
        .       .       .       .       .
```

Если EAX содержит нулевое значение, то команда CMP устанавливает флаг нуля ZF в единицу. Команда JZ проверяет флаг ZF и, если он равен 1, передает управление на адрес, указанный в ее операнде, то есть на метку NEXT. Фактиче-

ски данный фрагмент программного кода реализует логическую структуру if, анализирующую условие равенства EAX нулю.

Пример. Рассмотрим ситуацию выбора между двумя различными командами условного перехода в зависимости от того, проверяется результат операции над числами без знака или над числами со знаком. Предположим, что требуется перейти к метке BXM, если содержимое регистра BX имеет большее значение, чем содержимое регистра AX. Тогда надо использовать последовательность команд

```
CMP     BX, AX
JA      BXM
```

если операнды не имеют знака, и последовательность команд

```
CMP     BX, AX
JG      BXM
```

если они имеют знак.

6.5. Установка байта по условию

Совместно с командами условной передачи управления используется также группа команд SETx (set – устанавливать). Ее формат

```
SETx операнд
```

Здесь x – модификатор.

Команды устанавливают значение операнда (регистра или ячейки памяти длинной в байт) после проверки условия, задаваемого модификатором, аналогично тому, как это делается в командах условного перехода. Модификатор имеет те же значения, что и модификатор в командах условной передачи управления (за исключением CXZ и ECXZ). Операнд равен 0, если условие ложно, и равен 1, если условие истинно.

Пример. В следующем фрагменте байт по адресу RES будет установлен в 0:

```
MOV     AX, 0
CMP     AX, 1
SETE    RES
```

Команда SETx очень удобна при организации вычислений по условию. При этом можно избавиться от ненужных команд переходов, что дает выигрыш в быстродействии.

6.6. Пересылка по условию

Следующая группа команд, которую мы рассмотрим, включает команды CMOVx (conditional move – условная пересылка). Формат этой команды выглядит так:

```
CMOVx приемник, источник
```

Здесь x – модификатор с такими же значениями, как в предыдущей команде. Приемник может быть 16- или 32-разрядным регистром, а источник – 16- или 32-разрядным регистром или ячейкой памяти. Команда проверяет условие и, если оно выполняется, копирует содержимое источника в приемник. Если условие не выполняется, приемник остается без изменений.

Команда CMOVx весьма полезна при разработке быстрых алгоритмов и оптимизации ветвлений. Она позволяет избавиться от ненужных команд переходов.

Пример. Пусть требуется найти модуль (абсолютное значение) числа. Используя обычную команду условного перехода JGE, можно сделать это с помощью следующего программного кода:

```
.DATA
NUML DD –18
.CODE
.     .     .     .     .
MOV       EAX, NUML
CMP       EAX, 0
JGE       EXIT
NEG       EAX
EXIT:
.     .     .     .     .
```

Как видно из исходного текста, после команды CMP программный код разветвляется. Этого можно легко избежать, если использовать команду CMOVL. Более быстродействующий код выглядит так:

```
.DATA
NUML      DD    –18
.CODE
.     .     .     .     .
MOV       EAX, NUML
MOV       EDX, EAX
NEG       EDX
CMP       EAX, 0
CMOVL     EAX, EDX
```

6.7. Команды управления циклами

Команды управления циклами обеспечивают условные передачи управления при организации циклов. У микропроцессора регистр ECX служит счетчиком числа повторений циклов. Каждая команда управления циклами уменьшает содержимое регистра ECX на 1, а затем использует его новое значение для «принятия решения» о выполнении или не выполнении перехода.

Основная команда этой группы LOOP (повторять цикл до конца счетчика) имеет формат

LOOP близкая_метка

Запись операнда *близкая_метка* подчеркивает, что метка перехода должна находиться не далее –128 или +127 байтов от команды, следующей за LOOP.

Команда уменьшает содержимое регистра ECX на 1 и передает управление операнду *близкая_метка*, если содержимое регистра ECX не равно 0. Например, для стократного выполнения определенной группы команд можно воспользоваться следующей конструкцией:

```
        MOV     ECX, 100        ; Загрузить число повторений в ECX
START:
        .   .   .   .   .       ; Повторяемая группа команд
        LOOP    START           ; Если ECX не равен 0, то перейти к метке
                                ; START, в противном случае выйти из цикла
```

Команда LOOP завершает выполнение цикла только в том случае, если содержимое регистра ECX уменьшено до 0. Однако во многих приложениях требуются такие циклы, которые должны завершаться при выполнении определенных условий до того, как содержимое регистра ECX достигнет нуля. Такое альтернативное завершение цикла обеспечивается командой LOOPE (повторять цикл, если равно), имеющей синоним LOOPZ (повторять цикл, если нуль), и командой LOOPNE (повторять цикл, если не равно), имеющей синоним LOOPNZ (повторять цикл, пока не нуль).

Команда LOOPE:

– уменьшает содержимое регистра ECX на 1,

– осуществляет переход, если содержимое регистра ECX не равно 0 и флаг нуля ZF равен 1.

Таким образом, повторение цикла завершается, если:

– либо содержимое регистра ECX равно 0,

– либо флаг ZF равен 0,

– либо оба они равны 0.

Обычно команда LOOPE используется для поиска первого ненулевого результата в серии операций.

Команда LOOPNE:

– уменьшает содержимое регистра ECX на 1,

– осуществляет переход, если содержимое регистра ECX не равно 0 и флаг нуля ZF равен 0.

Таким образом, повторение цикла завершается, если:

– либо содержимое регистра ECX равно 0,

– либо флаг ZF равен 1,

– либо будет выполнено и то, и другое.

Обычно команда LOOPNE используется для поиска первого нулевого результата в серии операций.

Пример. В регистре EBX содержится адрес начала таблицы байтов.

```
        DEC     EBX         ; Предварительное уменьшение EBX
NEXT:
        INC     EBX         ; Передвинуть указатель к следующей
        CMP     [EBX], 0    ; ячейке и сравнить ее с 0
        LOOPE   NEXT        ; Перейти к сравнению следующего байта
        JNZ     NZ          ; Найден ненулевой байт?
.   .   .   .   .           ; Нет
```

```
NZ:                          ; Да
.    .    .    .    .
```

При создании вложенных циклов возникает проблема с содержимым регистра ECX, поскольку только он может использоваться в качестве счетчика цикла. Для временного сохранения счетчика внешнего цикла на время выполнения внутреннего доступно несколько способов: задействовать регистры, ячейки памяти или стек.

Пример. Будем сохранять счетчик внешнего цикла в ячейке памяти.

```
.DATA
     COUNT   DWORD       ?
. CODE
.    .    .    .    .
     MOV     ECX, 100    ; Установить счетчик внешнего цикла
L1:
     MOV     COUNT, ECX  ; Сохранить счетчик внешнего цикла
     MOV     ECX, 20     ; Установить счетчик внутреннего цикла
L2:
.    .    .    .    .
     LOOP    L2          ; Повторить внутренний цикл
     MOV     ECX, COUNT  ; Восстановить счетчик внешнего цикла
     LOOP    L1          ; Повторить внешний цикл
```

Таким образом, в теле внутреннего цикла доступны значения обоих счетчиков: счетчик внешнего цикла – в ячейке COUNT, счетчик внутреннего цикла – в регистре ECX.

7. Взаимодействие языков C++ и ассемблера

7.1. Использование подпрограмм на языке ассемблера в программах на языке C++

Поскольку писать на языке ассемблера достаточно объемные программы трудно, часто используется комбинирование программ на языке высокого уровня с кодом на ассемблере. Такой способ обычно используют в том случае, если в программе есть фрагменты, которые либо вообще невозможно реализовать без ассемблера, либо ассемблер может значительно повысить эффективность работы программы.

Заметим, что различные компиляторы языка C++ организуют такое взаимодействие не совсем одинаково. При работе с конкретным компилятором необходимо выяснять и учитывать имеющиеся особенности.

В среде Microsoft Visual C++ возможна разработка программ, разные модули которых написаны на разных языках: C/C++ и ассемблера. Для формирования такой программы необходимо создать проект, в который включить необходимые модули, написанные на языке C/C++ и языке ассемблера.

Изучая взаимодействие C++ и ассемблера, будем называть вызывающую функцию (на языке C++) или процедуру (на языке ассемблера) *программой*, а вызываемую функцию или процедуру – *подпрограммой.*

Сначала рассмотрим случай, когда программа на языке C++ вызывает подпрограмму на языке ассемблера.

7.1.1. Основы взаимодействия языков C++ и ассемблера

Первое требование, которое необходимо выполнить при совместной компоновке нескольких объектных файлов, созданных ассемблером и компилятором языка C++, состоит в том, что все объектные файлы должны иметь одинаковый формат – тот, который нужен загрузчику. Проблемы могут возникнуть при работе с объектными файлами, получаемыми с помощью средств разных производителей.

Другое требование – процедура на языке ассемблера должна учитывать соглашения по вызову функций, используемые компилятором языка C++. Для этого программист должен усвоить ряд деталей, которые можно почерпнуть либо из документации компилятора, либо анализируя машинный код, выдаваемый компилятором.

Необходимо учитывать следующие соглашения:

– подпрограмма не должна уничтожать регистровые значения программы (не обязательно все, так, большинство компиляторов C/C++ требуют сохранения только регистров EBX, ESI, EDI, ESP, EBP);

– программа и подпрограмма должны придерживаться общих соглашений о передаче данных из программы в подпрограмму и о возвращении данных из подпрограммы в программу.

7.1.2. Передача управления в подпрограмму и обратно

При вызове программой на языке C++ подпрограммы на языке ассемблера должны быть выполнены следующие шаги.

1. Программа должна сохранить адрес команды, с которой будет продолжено ее исполнение после завершения вызова подпрограммы. Затем программа передает управление подпрограмме.

2. После завершения подпрограмма должна возвратить управление по адресу, сохраненному ранее программой.

Реализация описанных шагов требует, чтобы программа сохранила адрес возврата в таком месте, которое доступно подпрограмме.

Для этого используется стек. Программа должна поместить адрес возврата в стек, а затем передать управление подпрограмме. Когда подпрограмма завершает работу, она извлекает из стека последний адрес возврата и возвращает управление этой ячейке памяти. Таким образом, механизм вызова подпрограмм един для ассемблера и C++.

Если разрабатываемая программа написана на C++ (файлы с текстами программы имеют расширения «cpp», а не «c», как для программ на Си), то в процес-

се компиляции происходит декорирование имен функций и переменных – добавление к имени символов, несущих дополнительную информацию о типах и пр. Подобная методика применяется в любом из языков высокого уровня, в которых поддерживается перегрузка имен функций. Чтобы не изучать правила декорирования, используемые конкретным компилятором языка C++, обычно пользуются правилами организации связи в стиле языка Си.

Для этого подпрограммы на языке ассемблера и переменные, используемые для передачи информации между программой и подпрограммой, в программе на языке C++ надо описывать как внешние (глобальные) с помощью модификатора extern "C".

Пример.

```
extern "C" void FUNC1( ); // объявить внешнюю функцию (процедуру)
void main( ) {
FUNC1( );
.       .       .       .       .
}
```

Встретив в программе на языке C++ инструкцию

```
FUNC1( );
```

компилятор сгенерирует машинную команду

```
CALL        _FUNC1
```

Символическое имя _FUNC1 соответствует глобальному идентификатору, определенному в подпрограмме; компилятор C++ добавляет знак подчеркивания ко всем глобальным идентификаторам (за некоторым исключением, о котором будет сказано ниже).

Пример. Приведем пример модуля, содержащего подпрограмму на языке ассемблера.

```
.386
.MODEL    FLAT
.CODE
_FUNC1    PROC
; Инструкции языка ассемблера
              RET
_FUNC1    ENDP
END
```

Если же в директиве MODEL для параметра язык указать значение C, то подчеркивание глобальных идентификаторов не потребуется:

```
.386
.MODEL    FLAT, C
.CODE
FUNC1     PROC
; Инструкции языка ассемблера
              RET
FUNC1     ENDP
END
```

Далее рассмотрим способы обмена данными между программой на языке C++ и подпрограммой на языке ассемблера. Существует два способа обмена данными: использование глобальных данных и аргументов. Рассмотрим каждый из них отдельно.

7.1.3. Использование глобальных переменных для передачи данных

Подпрограмма на языке ассемблера может иметь доступ к глобальным переменным, которые определены в модуле на языке C++ с использованием внешнего класса хранения данных. Напомним, что внешними объектами в программе на языке C++ являются те, которые определены вне любой функции и не объявлены ключевым словом static.

Чтобы обеспечить подпрограмме на языке ассемблера доступ к внешним объектам, в ней надо использовать для ссылки на внешние объекты директиву EXTERN (или EXTRN).

Пример. Подпрограмма изменяет значение глобальной переменной VAL.

Модуль на языке C++.

```
# include "stdlib.h"
extern "C" void COPY( );
extern "C" int VAL = 0;  // определить внешний объект данных,
                         // инициализация обязательна
main ( ) {
  COPY( );
  printf ("VAL = %d\n", VAL);
}
```

Модуль на языке ассемблера.

```
.386
.MODEL    FLAT
EXTERN    _VAL: DWORD   ; Имя _VAL объявлено как глобальный
                        ; объект длиной в двойное слово
.CODE
_COPY     PROC
          MOV       EAX, 4
          MOV       _VAL, EAX ; Изменяем глобальный объект
          RET
_COPY     ENDP
END
```

С другой стороны можно получить доступ из программы на языке C++ к переменным, описанным в модуле на языке ассемблера. Для этого нужные переменные в модуле на языке C++ необходимо описать как внешние объекты, а в модуле на языке ассемблера – как общедоступные с помощью директивы PUBLIC.

Пример. Модифицируем предыдущий пример.

Модуль на языке C++.

```
# include "stdlib.h"
extern "C"  void COPY( );
```

```
extern "C" int VAL;        // объявить внешний объект данных,
                           // теперь без инициализации
main ( ) {
  COPY( );
  printf ("VAL = %d\n", VAL);
}
```

Модуль на языке ассемблера.

```
.386
.MODEL    FLAT
PUBLIC    _VAL; Имя _VAL объявлено как общедоступный объект
.DATA
_VAL      DD    ?
.CODE
_COPY     PROC
          MOV   EAX, 4
          MOV   _VAL, EAX ; Изменяем общедоступный объект
          RET
_COPY     ENDP
END
```

7.1.4. Использование аргументов для передачи данных

Передача данных может осуществляться через *аргументы*. При передаче аргументов Visual C++ расширяет их до 32 бит, если они имеют меньший размер.

В языке C++ используется несколько разных соглашений о вызовах, определяющих, в частности, способ передачи аргументов.

По умолчанию в языке C++ используется соглашение, называемое *cdecl* (им мы до сих пор и пользовались). Согласно этому соглашению аргументы передаются по значению, т.е. подпрограмма получает копию каждого аргумента. Аргументы помещаются в стек в порядке, *обратном* тому, в котором они указаны при вызове процедуры. По окончании выполнения подпрограммы программа очищает стек от аргументов.

Пример. Если A1, A2, A3 – переменные типа int, то вызов процедуры FUNC(Al, A2, A3) преобразуется компилятором C++ в последовательность команд

```
PUSH      A3
PUSH      A2
PUSH      A1
CALL_FUNC
ADD       ESP, 12
```

При входе в подпрограмму ячейка стека с адресом возврата имеет адрес, хранящийся в ESP. Тогда к копиям аргументов в стеке можно обратиться с использованием адресации по базе: [ESP+4], [ESP+8], [ESP+12].

Общепринятым, однако, считается обращение к аргументам в стеке через регистр указателя базы EBP. Поэтому одной из первых команд любой подпро-

граммы, которой необходимо адресоваться к аргументам, является помещение в регистр EBP значения регистра ESP. Однако по соглашению нельзя уничтожать содержимое регистра EBP в подпрограмме. Поэтому следует сначала сохранить содержимое регистра EBP, а затем воспользоваться регистром EBP в подпрограмме. Обычно регистры сохраняются в стеке. Поэтому сначала необходимо сохранить значение EBP в стеке, а затем воспользоваться регистром EBP в подпрограмме; непосредственно перед возвратом в программу нужно восстановить исходное значение EBP.

Таким образом, общепринятые заголовок (пролог) и заключительная часть (эпилог) подпрограммы имеют следующий вид:

```
_FUNC       PROC
    PUSH    EBP         ; Сохранить значение EBP при вызове
    MOV     EBP, ESP    ; Установить новое значение EBP
      .     .     .     .     .
    POP     EBP         ; Восстановить исходное значение EBP
    RET                 ; Вернуться в вызывающую функцию
_FUNC   ENDP
```

В этом случае на время исполнения подпрограммы стек будет содержать не только аргументы и адрес возврата, но еще и сохраненное значение регистра EBP.

Пример. Используем тот же вызов функции, что и в предыдущем примере:

FUNC(Al, A2, A3);

После исполнения первых двух команд подпрограммы (PUSH и MOV) адреса аргументов будут следующими (см. рисунок 17):

EBP+8 – адрес первого аргумента,

EBP+12 – адрес второго аргумента,

EBP+16 – адрес третьего аргумента.

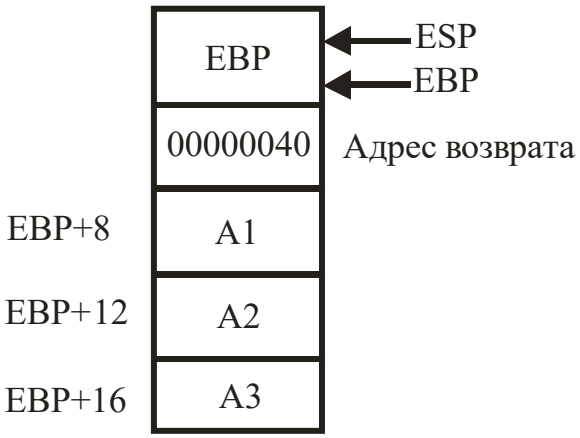

Рисунок 17.

Заметим, что на первый взгляд непосредственное использование регистра ESP для доступа к аргументам кажется более простым. Однако, в этом случае программисту пришлось бы отслеживать использование команд, изменяющих значение регистра ESP, а последнее при регулярной (а, тем более, эпизодической)

модификации программы, является слишком сложным. В результате такой способ чреват ошибками.

Пример. Подпрограмма получает три целых аргумента, суммирует их, а затем возвращает полученный результат через внешнюю переменную VAL.

Модуль на языке C++.

```
# include "stdlib.h"
extern "C" void FUNC(int A1, int A2, int A3);
extern "C" int VAL = 0;
void main( ){
int x = 10;
FUNC(x, 20, 20+5);
printf("VAL = %d\n", VAL);
}
```

Модуль на языке ассемблера.

```
.386
.MODEL      FLAT
EXTERN      _VAL: DWORD
.CODE
_FUNC       PROC
        PUSH    EBP             ; Сохранить значение EBP при вызове
        MOV     EBP, ESP        ; Установить новое значение EBP
        MOV     EAX, [EBP+8]    ; Первый аргумент поместить в EAX
        ADD     EAX, [EBP+12]   ; Добавить значение второго аргумента
        ADD     EAX, [EBP+16]   ; Добавить значение третьего аргумента
        MOV     _VAL, EAX       ; Поместить результат в _VAL
        POP     EBP             ; Восстановить исходное значение EBP
        RET
_FUNC       ENDP
END
```

В подпрограмме на языке ассемблера первый аргумент загружается в регистр EAX, а значения остальных аргументов добавляются к этому регистру с помощью команды сложения ADD. Затем результат (находящийся в регистре EAX) помещается во внешний объект VAL.

Помимо соглашения cdecl имеется еще ряд соглашений о вызовах, некоторые из них описаны в таблице 11.

Таблица 11

Директива	Порядок передачи параметров	Очистка стека	Использование регистров	Комментарий
cdecl	обратный	программа	нет	по умолчанию в C/C++
pascal	прямой	подпрограмма	нет	в Visual C++ не

				используется
stdcall	обратный	подпрограмма	нет	для функций API
fastcall	прямой/ обратный	подпрограмма	ECX, EDX	самый быстрый

Соглашение *pascal* языка Паскаль является противоположным по отношению к cdecl. Однако, поскольку это соглашение в Visual C++ не используется, примеры его применения рассматривать не будем.

Рассмотрим соглашение *stdcall*, согласно которому аргументы помещаются в стек в обратном порядке, но освобождает стек от помещенных в него аргументов подпрограмма.

В этом случае описание подпрограммы в модуле на C++ выглядит следующим образом:

```
extern "C" int __stdcall FUNC(int A1, int A2, int A3); // используется
                                                       // двойное подчеркивание
```

В модуле на языке Ассемблера подпрограмма должна иметь имя _FUNC@12 (после знака @ указывается общая длина передаваемых аргументов), а команда RET должна иметь аргумент, равный этой длине.

Пример. Модифицируем предыдущий пример.

Модуль на языке C++.

```
# include "stdlib.h"
extern "C" void __stdcall FUNC(int A1, int A2, int A3);
extern "C" int VAL = 0;
void main( ){
int x = 10;
FUNC(x, 20, 20+5);
printf("VAL = %d\n", VAL);
}
```

Модуль на языке ассемблера.

```
.386
.MODEL      FLAT
EXTERN      _VAL: DWORD
.CODE
_FUNC@12    PROC
        PUSH    EBP             ; Сохранить значение EBP при вызове
        MOV     EBP, ESP        ; Установить новое значение EBP
        MOV     EAX, [EBP+8]    ; Первый аргумент поместить в EAX
        ADD     EAX, [EBP+12]   ; Добавить значение второго аргумента
        ADD     EAX, [EBP+16]   ; Добавить значение третьего аргумента
        MOV     _VAL, EAX       ; Поместить результат в _VAL
        POP     EBP             ; Восстановить исходное значение EBP
        RET     12              ; Аргументы занимают в стеке 12 байтов
_FUNC@12    ENDP
END
```

Еще одно соглашение – *fastcall* – использует самый быстрый способ передачи аргументов – через регистры. Согласно ему первый и второй аргументы передаются через регистры ECX и EDX соответственно. Остальные аргументы передаются через стек в обратном порядке. Освобождает стек от помещенных в него аргументов подпрограмма.

Заметим, что передача аргументов через регистры общего назначения часто используется и в программах на языке ассемблера. Однако, главное ограничение здесь – малое число регистров микропроцессора. Кроме того, обычно при загрузке в регистры значений аргументов приходится сохранять в стеке их текущее состояние.

Описание подпрограммы в модуле на C++ при использовании соглашения fastcall выглядит следующим образом:

```
extern "C" void __fastcall FUNC(int A1, int A2, int A3); // используется
                                                         // двойное подчеркивание
```

В модуле на языке Ассемблера подпрограмма FUNC должна иметь имя @FUNC@12 (знак подчеркавания не используется), команда RET должна иметь аргумент 4 (через стек передан только один аргумент).

Пример. Продолжим модификацию нашего примера.

Модуль на языке C++.

```
# include "stdlib.h"
extern "C" void __fastcall FUNC(int A1, int A2, int A3);
extern "C" int VAL = 0;
void main( ){
int x = 10;
FUNC(x, 20, 20+5);
printf("VAL = %d\n", VAL);
}
```

Модуль на языке ассемблера.

```
.386
.MODEL      FLAT;
EXTERN      _VAL: DWORD
.CODE
@FUNC@12    PROC
        PUSH    EBP             ; Сохранить значение EBP при вызове
        MOV     EBP, ESP        ; Установить новое значение EBP
        MOV     EAX, ECX        ; Первый аргумент поместить в EAX
        ADD     EAX, EDX        ; Добавить значение второго аргумента
        ADD     EAX, [EBP+8]    ; Добавить значение третьего аргумента
        MOV     _VAL, EAX       ; Поместить результат в _VAL
        POP     EBP             ; Восстановить исходное значение EBP
        RET     4
@FUNC@12    ENDP
END
```

Наряду с возвращением значений через глобальные объекты подпрограмма на языке ассемблера может возвращать их через аргументы вызова или как значе-

ние подпрограммы. Далее рассмотрим реализацию этих способов возвращения значений.

7.1.5. Возвращение значения через имя подпрограммы

Если подпрограмма возвращает только одно значение, то проще всего трактовать его как значение имени подпрограммы. Чтобы вызывающая программа могла воспринять возвращаемую сумму как значение имени подпрограммы, нужно следующим образом модифицировать наш пример (используется соглашение cdecl).

Модуль на языке C++.

```
# include "stdlib.h"
extern "C" int FUNC(int A1, int A2, int A3);
void main( ){
int val, x = 10;
val = FUNC(x, 20, 20+5);
printf("val = %d\n , val);
}
```

Программа и подпрограмма должны придерживаться определенных соглашений относительно возвращения значения через имя подпрограммы. Обычно в компиляторах языка C++ предполагается, что это значение запомнено в одном или нескольких регистрах.

В Visual C++ возвращаемые значения расширяются до 32 битов, если имеют меньший размер, и возвращаются в регистре EAX (в результате однобайтовый объект возвращается в регистре AL, а двухбайтовый – в регистре AX). Элементы длиной 64 бита возвращаются в паре регистров EDX:EAX. Структуры большего размера возвращаются через указатели в регистре EAX на скрытые возвращаемые данные.

Пример. Чтобы в соответствии с этими соглашениями модифицировать подпрограмму FUNC, надо просто оставить вычисленный ею результат в регистре EAX, а также удалить ссылки на внешний объект VAL.

Программа на языке C++ извлечет результат из регистра EAX, когда ей будет возвращено управление.

Модуль на языке ассемблера.

```
.386
.MODEL      FLAT
.CODE
_FUNC       PROC
        PUSH    EBP             ; Сохранить значение EBP при вызове
        MOV     EBP, ESP        ; Установить новое значение EBP
        MOV     EAX, [EBP+8]    ; Первый аргумент поместить в EAX
        ADD     EAX, [EBP+12]   ; Добавить значение второго аргумента
        ADD     EAX, [EBP+16]   ; Добавить значение третьего аргумента
        POP     EBP             ; Восстановить исходное значение EBP
```

```
        RET
_FUNC           ENDP
END
```

7.1.6. Использование аргументов для возвращения значений

Если подпрограмма должна возвращать несколько значений, то она может использовать для этой цели свои аргументы. На языке C++ это производится путем передачи аргументов *по адресу*. Доступ к аргументам, передаваемым по адресу, можно получить и в подпрограммах на языке ассемблера.

Еще раз модифицируем подпрограмму FUNC: теперь у нее будет четыре аргумента, и результат будет возвращаться через последний из них – локальную переменную val. Вызов подпрограммы будет иметь следующий вид:

```
FUNC(x, 20, 20+5, &val);
```

После входа в подпрограмму и инициализации регистра EBP адрес четвертого аргумента будет EBP+20. После загрузки четвертого аргумента в регистр общего назначения мы сможем поместить по адресу val значение, используя косвенную адресацию.

```
MOV     EBX, [EBP+20] ; Загрузить адрес, по которому должен быть
                      ; помещен результат
MOV     [EBX], EAX    ; Поместить значение суммы по этому адресу
```

7.2. Вызов подпрограмм на языке C++ из программ на языке ассемблера

Чаще всего приходится вызывать подпрограммы на языке ассемблера из программ на языке C++. Однако можно вызывать и подпрограммы на языке C++ из программ (или подпрограмм) на языке ассемблера. Далее будем использовать соглашения cdecl.

Для вызова подпрограммы на языке C++ из программы на языке ассемблера надо объявить имя подпрограммы в программе путем использования директивы EXTERN. Если имя подпрограммы на языке C++ – CFUNC, то ее объявление будет иметь вид

```
EXTERN    _CFUNC: PROC
```

Чтобы вызвать подпрограмму на языке C++, каждый ее аргумент нужно поместить в стек, начиная с последнего, а затем вызвать подпрограмму с помощью команды CALL. После возвращения из подпрограммы на языке C++ программа на языке ассемблера должна очистить стек, удалив из него все ранее помещенные аргументы. Для этого можно с помощью команды POP извлечь их один за другим. Но легче всего просто увеличить содержимое указателя стека ESP на целое значение, которое равно числу байтов, ранее помещенных в стек.

Если данные определены внутри модуля на языке ассемблера, то их можно сделать доступными в модуле на языке C++, объявив общедоступными с помощью директивы PUBLIC в модуле на языке ассемблера и внешними с помощью описания extern "C" в модуле на языке C++.

Пример. Вызов подпрограммы на языке C++ из программы на языке ассемблера.

Модуль на языке C++.

```
extern "C" int SUM; // переменная описана в модуле на языке ассемблера
extern "C" void CFUNC(int a, int b) // используется соглашение cdecl
{
        SUM = a+b;
}
```

Модуль на языке ассемблера.

```
.386
.MODEL      FLAT
PUBLIC      _SUM
EXTERN      _CFUNC: PROC
.DATA
_SUM        DD    ?
.CODE
_ASMPROG    PROC
    PUSH    3
    PUSH    5
    CALL    _CFUNC
    ADD     ESP, 8
.   .   .   .   .
    RET
_ASMPROG    ENDP
END         _ASMPROG
```

В этом примере перед выполнением команды CALL выполняются две команды PUSH, в результате чего содержимое указателя стека уменьшается на восемь байтов. Увеличение указатель стека ESP на восемь после выполнения команды CALL позволяет восстановить его исходное содержимое.

7.3. Использование локальных данных

Для хранения автоматических объектов данных в функциях на языке C++ используется стек. Это делается путем уменьшения содержимого указателя стека ESP на число байтов, занимаемых объектом данного типа.

Например, если в функции определены три автоматических переменных типа int, то после команды сохранения значения указателя базы EBP при входе в функцию компилятор языка C++ должен генерировать команду

```
SUB         ESP, 12
```

Эта команда выделит в стеке 12 байтов, для доступа к которым используются регистр EBP и адресация по базе.

Перед возвращением в вызывающую функцию надо очистить стек, восстановив содержимое указателя стека.

Рассмотрим два распространенных варианта работы со стеком.

Вариант 1 (см. рисунок 18)

```
PUSH    EBP         ; Сохранить значение EBP при вызове
MOV     EBP, ESP    ; Установить новое значение EBP
SUB     ESP, n      ; Выделить n байтов
.   .   .   .   .
MOV     ESP, EBP    ; Освободить память
POP  EBP            ; Восстановить EBP
RET
```

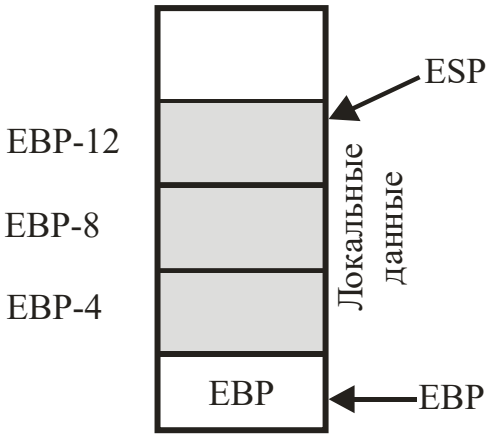

Рисунок 18.

Вариант 2 (см. рисунок 19)

```
PUSH    EBP         ; Сохранить значение EBP при вызове
SUB     ESP, n      ; Выделить n байтов
MOV     EBP, ESP    ; Установить новое значение EBP
.   .   .   .   .
ADD     ESP, n      ; Освободить память
POP     EBP         ; Восстановить EBP
RET
```

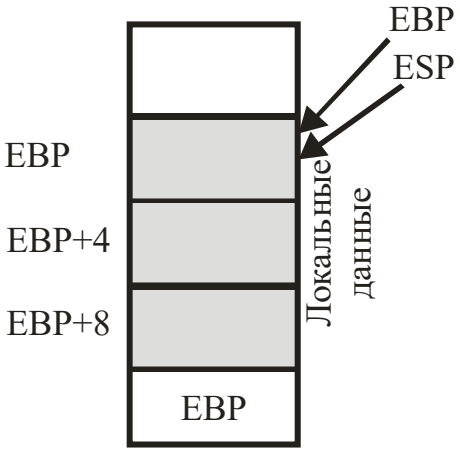

Рисунок 19.

Обратим внимание на то, что в первом варианте указатель базы EBP устанавливается до выделения в стеке памяти для локальных данных, а во втором – после выделения памяти.

В первом варианте указатель базы EBP продолжает показывать на ту ячейку стека, на которую показывал указатель стека ESP до выделения памяти для локальных данных. Поэтому перед выполнением команды RET стек можно очистить от этих данных путем присвоения указателю стека ESP значения указателя базы EBP. Во втором варианте стек очищается увеличением содержимого указателя стека ESP на то же значение, на которое оно было уменьшено при выделении памяти для локальных данных.

Совместное расположение в стеке параметров подпрограммы и локальных переменных иллюстрирует рисунок 20. Здесь используется первый варианта работы со стеком, что позволяет обращаться к аргументам независимо от количества локальных данных и, вообще, от их наличия. Заметим, что область памяти, изображенная на рисунке 20, называется стековым кадром или фреймом (к нему относятся также другие сохраненные в начале работы функции регистры).

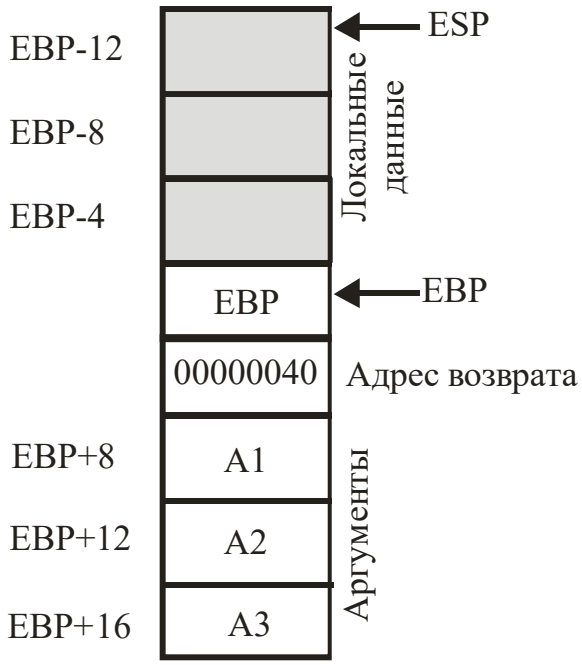

Рисунок 20.

7.4. Использование библиотечных функций языка Си в программах/подпрограммах на языке ассемблера

В подпрограммах и программах на языке ассемблера, разрабатываемых в среде Visual C++, можно использовать функции стандартной библиотеки языка Си.

В подпрограммах на языке ассемблера, вызываемых из программы на языке C++, а также в программах на языке ассемблера, вызывающих подпрограммы на

языке C++, нужные библиотечные функции достаточно описать директивой EXTERN. Используемое при этом соглашение о вызовах – cdecl.

Пример. Подпрограмма на языке ассемблера выдает приглашение на ввод строки с использованием функции printf, осуществляет ввод строки с использованием функции scanf и передает указатель на нее в вызывающую программу.

```
.386
.MODEL      FLAT
EXTERN      _scanf: PROC ; Описание используемых библиотечных
EXTERN      _printf: PROC ; функций языка Си
.DATA
STRN        DB    "Enter string",13,10,0
FMT         DB    "%s",0
BUF         DB    200 DUP (?)
.CODE
_ASMIO      PROC
        PUSH    OFFSET STRN
        CALL    _printf
        ADD     ESP, 4
        PUSH    OFFSET BUF
        PUSH    OFFSET FMT
        CALL    _scanf
        ADD     ESP, 8
        MOV     EAX, OFFSET BUF
        RET
_ASMIO      ENDP
END
```

Заметим, что вместо управляющих последовательностей форматированного ввода/вывода языка Си в форматной строке необходимо указывать соответствующие ASCII коды. Например, последовательность «13,10» в строке STRN, является аналогом последовательности «\n». Приведем также пример использования табуляции «\t» (ASCII код 9) при выводе трех целых чисел:

```
FMT1DB    "%d",9,"%d",9,"%d",13,10,0
```

В программах, использующих только язык ассемблера, также можно использовать функции стандартной библиотеки языка Си (в среде Visual C++). Однако, для этого необходимо подключить подходящий вариант библиотеки с помощью директивы INCLUDELIB.

Пример. Модифицируем предыдущий пример. В конце программы на языке ассемблера введенная строка сразу выводится на экран.

```
.386
.MODEL      FLAT
INCLUDELIB  MSVCRTD.LIB ; Подключение библиотеки
EXTERN      _scanf: PROC
EXTERN      _printf: PROC
.DATA
STRN        DB    "Enter string",13,10,0
```

```
FMT             DB    "%s",0
BUF             DB    200 DUP (?);
.CODE
_ASMIO          PROC
        PUSH    OFFSET STRN
        CALL    _printf
        ADD     ESP, 4
        PUSH    OFFSET BUF
        PUSH    OFFSET FMT
        CALL    _scanf
        ADD     ESP, 4
        CALL    _printf
        ADD     ESP, 4
        RET
_ASMIO          ENDP
END             _ASMIO
```

Обратим внимание, что после вызова функции scanf увеличением ESP на 4 стек освобождается только от адреса строки FMT, адрес строки BUF остается в стеке, поскольку он необходим как аргумент функции printf, после вызова последней увеличением ESP на 4 стек освобождается и от адреса строки BUF.

7.5. Использование вставок на языке ассемблера в программах на языке C++

Многие компиляторы языков высокого уровня, в том числе C/C++, поддерживают возможность непосредственного размещения в программе на этом языке кода, написанного на языке ассемблера.

Использование встроенного ассемблера является хорошей альтернативой написанию внешних модулей на ассемблере. Основное преимущество здесь заключается в простоте написания ассемблерных вставок, поскольку не нужно учитывать особенности компоновки объектных модулей, именования внешних идентификаторов и порядок передачи параметров в процедуры. Основной недостаток использования встроенного ассемблера состоит в том, что в результате получается непереносимый код программы. Переносимость программы важна в случае, если она должна компилироваться под разные платформы.

В Visual C++ директиву __asm (перед словом «asm» указываются два символа подчеркивания) можно использовать двояко. Во-первых, ее можно поместить в начало строки перед одиночной ассемблерной командой:

```
__asm команда
```

Во-вторых, можно создать блок ассемблерных команд (он называется asm-блоком):

```
__asm
{
        команда-1
        команда-2
.       .       .       .       .
```

```
        команда-n
}
```

В любое место ассемблерного блока можно помещать комментарии после любой из его команд, точно так же, как это делается в любой ассемблерной программе. При этом можно пользоваться синтаксисом комментариев, принятых либо в языке ассемблера, либо в C/C++.

Рассмотрим основные возможности, которые может использовать программист при написании ассемблерных вставок:

– использовать в программе любой регистр, предусмотренный в структуре микропроцессора;

– использовать в качестве операнда имя регистра;

– обращаться к параметру функции по имени;

– обращаться к меткам и переменным, которые были объявлены за пределами ассемблерного блока (этот момент хочется подчеркнуть особо, поскольку локальные переменные функции должны быть объявлены за пределами ассемблерного блока);

– использовать числовые литералы, заданные в стиле либо языка ассемблера, либо языка Си (например, литералы 0A26H и 0xA26 эквивалентны и могут совершенно свободно использоваться в программе);

– использовать операцию PTR в командах типа

```
INC   BYTE PTR [ESI].
```

В тоже время при написании ассемблерных вставок имеется ряд ограничений. В частности, нельзя использовать директивы определения данных, такие как DB и DW.

В начале выполнения ассемблерного блока содержимое регистров общего назначения не определено. На этот счет нельзя строить каких-либо предположений, поскольку значение регистра будет зависеть от выполняемого кода перед ассемблерным блоком.

Во встроенном ассемблерном блоке можно без всяких ограничений использовать регистры EAX, EBX, ECX, EDX, ESI, EDI, поскольку компилятор не сохраняет содержимое регистров при переходе от одного оператора к другому и поэтому не учитывает значения, оставшиеся в регистрах с момента выполнения предыдущего оператора (однако, необходима осторожность для функций, объявленных с атрибутом __fastcall, поскольку в этом случае в некоторых регистрах хранятся аргументы). С другой стороны, если вы измените содержимое всех регистров, то компилятор C++ не сможет выполнить полную оптимизацию кода вашей процедуры, поскольку для этого ему нужны свободные регистры.

В ассемблерном блоке нельзя воспользоваться операцией OFFSET, однако существует возможность загрузить адрес переменной с помощью команды LEA. Например, в приведенной ниже команде в регистр ESI загружается адрес переменной BUFFER:

```
LEA   ESI, BUFFER
```

8. Команды манипулирования битами

Данные команды манипулируют отдельными битами или группами битов в регистрах или ячейках памяти.

8.1. Логические команды AND, OR, XOR и NOT

Основные логические команды выполняют над операндами операции логического умножения AND (И), логического сложения OR (ИЛИ), логического исключающего сложения XOR (Исключающее ИЛИ) и отрицания NOT (НЕ). Операции выполняются побитно.

Формат команд AND, OR и XOR

AND приемник, источник

OR приемник, источник

XOR приемник, источник

Операндами команд AND, OR и XOR могут быть байты, слова или двойные слова. В этих командах можно сочетать два регистра, регистр с ячейкой памяти или непосредственное значение с регистром или ячейкой памяти.

Команды всегда сбрасывают флаги переполнения (OF) и переноса (CF). Кроме того, они устанавливают значения флагов знака (SF), нуля (ZF) и четности (PF) в соответствии со значением результата.

Команда NOT обращает состояние каждого бита регистра или ячейки памяти (своего единственного операнда) и ни на какие флаги не воздействует.

Примеры.

```
AND     AX, BX
OR      AL, MEM_BYTE
XOR     EAX, EAX ; Эффективный способ обнуления регистра
NOT     AX
```

Команда AND обычно используется для сброса отдельных битов двоичного числа (например, флагов состояния процессора) по заданной маске. Если бит маски равен 1, значение соответствующего разряда числа не изменяется (в этом случае говорят, что разряд маскирован), а если равен 0 – то сбрасывается. В качестве примера сбросим четыре старших бита 8-разрядного двоичного числа. Для выполнения этой операции можно воспользоваться двумя командами:

```
MOV AL, 00111011B
AND AL, 00001111B
```

Команда AND позволяет очень просто преобразовать строчные латинские буквы в прописные. Действительно, сравнив двоичные ASCII-коды прописной «A» и строчной «a», можно заметить, что они отличаются только значением 5-го разряда:

01100001 = 61H (a)

01000001 = 41H (A)

Остальные символы упорядочены в алфавитном порядке, но для них выполняется то же правило. Следовательно, если значение маски выбрать равным 11011111B, то при выполнении команды AND мы сбросим только значение 5-го бита числа, оставив все остальные биты без изменений.

Команда OR обычно используется для установки в единицу отдельных битов двоичного числа (например, флагов состояния процессора) по заданной маске. Если бит маски равен 0, значение соответствующего разряда числа не изменяется, а если равен 1 – то устанавливается в 1.

С помощью команды OR можно преобразовать двоичное число, значение которого находится в диапазоне от 0 (00000000B) до 9 (00001001B) в соответствующий ему ASCII-код. Для этого необходимо к нужному числу прибавить 30H=00110000B, т.е. установить в единицу биты 4 и 5. Например, если в регистре AL находится число 09H, то чтобы преобразовать его в соответствующий ASCII-код, нужно выполнить операцию OR регистра AL с числом 30H. На языке ассемблера подобное преобразование можно записать так:

```
MOV     DL, 00001001B; 9H
OR      DL, 00110000B; 30H
```

Результат в регистре DL: 00111001B = 39H.

С помощью команды OR можно определить, какое значение находится в регистре (отрицательное, положительное или нуль). Для этого вначале нужно выполнить команду OR, указав в качестве операндов один и тот же регистр, например:

```
OR      AL, AL
```

а затем – проанализировать значения флагов, как показано в таблице 12.

Таблица 12

Флаг нуля (ZF)	Флаг знака (SF)	Значение числа
0	0	Больше нуля
1	0	Равно нулю
0	1	Меньше нуля

Команда XOR применяется для выяснения того, какие биты в обоих операндах различаются, или для инвертирования заданных битов в первом операнде. Интересующие нас биты маски (второго операнда) при выполнении команды XOR должны быть единичными, остальные – нулевыми:

```
XOR     EAX, 10B    ; Инвертировать 1-й бит в регистре EAX
```

Команда XOR обладает свойством реверсивности – если ее выполнить дважды с одними и теми же операндами, то восстановится исходное значение результата. Так, если два раза подряд выполнить операцию XOR между битами A и B, то в результате получится исходное значение бита A.

Пример. Воспользуемся свойством реверсивности операции исключающего ИЛИ для выполнения простого шифрования данных.

В процессе шифрования исходная строка, введенная пользователем с клавиатуры (назовем ее открытым текстом), преобразовывается в непонятный набор байтов (назовем его зашифрованным текстом) с помощью другой строки, называемой ключом. Зашифрованный текст можно сохранять или передавать адресату, не опасаясь, что кто-то посторонний сможет его прочитать. Получив зашифро-

ванный текст, авторизованный пользователь после применения программы дешифрования сможет восстановить первоначальное сообщение (т.е. снова получить открытый текст). Мы воспользуемся так называемым методом симметричного шифрования, означающим, что для шифрования и последующей расшифровки используется один и тот же ключ.

Приведем фрагмент, осуществляющий шифрование/дешифрование строки байтов BUF, имеющей длину LEN, ключ имеет значение CHAR.

```
        MOV     ESI, OFFSET BUF
        MOV     ECX, LEN
        MOV     AL, CHAR
L1:
        XOR     [ESI], AL
        INC     ESI
        LOOP    L1
```

8.2. Команда проверки TEST

Команда TEST (проверить) выполняет операцию AND над операндами, но воздействует только на флаги и не изменяет значения операндов.

Команда TEST обычно используется совместно с идущей вслед за ней командой условного перехода.

С помощью команды TEST можно определить состояние заданных битов в первом операнде. Проверяемые биты первого операнда в маске (втором операнде) должны иметь единичное значение.

Результатом команды является установка значения флага нуля ZF:

– если ZF = 0, то в результате логического умножения получился ненулевой результат, то есть хотя бы один единичный бит маски совпал с соответствующим единичным битом первого операнда;

– если ZF = 1, то в результате логического умножения получился нулевой результат, то есть ни один единичный бит маски не совпал с соответствующим единичным битом первого операнда.

Например,

```
        TEST    EAX, 00000010H
        JNZ     ML ; Переход если 4-й бит равен 1
```

8.3. Команды сканирования битов

Данные две команды позволяют осуществить поиск первого установленного в 1 бита операнда. Поиск можно произвести как с начала, так и от конца операнда.

Команда BSF (bit scanning forward – сканирование битов вперед) имеет формат

BSF приемник, источник

Команда просматривает (сканирует) биты операнда-источника от младшего к старшему (от бита 0 до старшего бита) в поисках первого бита, установленного в 1. Если таковой обнаруживается, в операнд-приемник заносится номер этого бита в виде целочисленного значения. Если все биты операнда-источника равны 0, то флаг нуля ZF устанавливается в 1, в противном случае флаг ZF сбрасывается в 0.

```
MOV     AL, 02H
BSF     BX, AL      ; BX=1
JZ      ML          ; Переход, если в AL нуль
.   .   .   .   .
```

Команда BSR (bit scanning reset – сканирование битов в обратном порядке) имеет формат

BSR приемник, источник

Команда просматривает (сканирует) биты операнда-источника от старшего к младшему (от старшего бита к биту 0) в поисках первого бита, установленного в 1. Если таковой обнаруживается, в операнд-приемник заносится номер этого бита в виде целочисленного значения. При этом важно, что позиция первого единичного бита слева все равно отсчитывается относительно бита 0. Если все биты операнда-источника равны 0, то флаг нуля ZF устанавливается в 1, в противном случае флаг ZF сбрасывается в 0.

8.4. Команды проверки и модификации битов

В данных командах операнд может находиться как в памяти, так и в регистре общего назначения. Положение бита задается его смещением относительно младшего бита операнда. Смещение может как задаваться в виде непосредственного значения, так и содержаться в регистре общего назначения. В качестве значения смещения можно использовать результаты работы команд BSR и BSF. Все команды присваивают значение выбранного бита флагу CF.

Команда BT (bit test – проверка бита) переносит значение бита во флаг CF, формат команды

BT операнд, смещение

Например:

```
BT    AX, 5 ; Проверить значение бита 5
JNC   ML    ; Переход, если бит равен нулю
```

Команда BTS (bit test and set – проверка и установка бита) переносит значение бита во флаг CF и затем устанавливает проверяемый бит в 1, формат команды

BTS операнд, смещение

Например:

```
MOV AX, 10
BTS   POLE, AX   ; Проверить и установить  бит 10
JC    ML         ; Переход, если проверяемый бит был равен 1
```

Команда BTR (bit test and reset – проверка и сброс бита) переносит значение бита во флаг CF и затем устанавливает этот бит в 0, формат команды

BTR операнд, смещение

Команда BTC (bit test and convert – проверка и инвертирование бита) переносит значение бита во флаг CF и затем инвертирует значение этого бита, формат команды

BTC операнд, смещение

8.5. Команды сдвига и циклического сдвига

У микропроцессора имеются команды, осуществляющие сдвиг 8-, 16- или 32-битового содержимого регистров или ячеек памяти на одну или несколько позиций влево или вправо.

Для всех команд флаг переноса CF является (в некотором смысле) расширением операнда битом 8, битом 16 или битом 32. Иначе говоря, флаг CF приобретает значение бита, сдвинутого за один из концов операнда.

Команды сдвига и циклического сдвига вправо помещают во флаг CF значение нулевого бита.

Команды сдвига и циклического сдвига влево помещают в него значение бита 7 (при операциях над байтом), бита 15 (при операциях над словом) или бита 31 (при операциях над двойным словом).

Команды сдвига и циклического сдвига имеют два операнда: приемник и счетчик.

Приемником может быть 8-, 16- или 32-битовый регистр общего назначения или ячейка памяти. Счетчик может быть числом или значением в регистре CL (воспринимаются только 5 младших битов, т.е. допустимы значения счетчика 0..31).

В последних моделях микропроцессора есть дополнительные команды, позволяющие делать 64-разрядные сдвиги.

8.5.1. Команды сдвига

Команды сдвига (см. рисунок 21) распадаются на две группы.

1. Логические команды сдвигают операнд, не считаясь с его знаком. Они используются для действий над числами без знака или над нечисловыми значениями,

2. Арифметические команды сохраняют старший, знаковый бит операнда. Они используются для действий над числами со знаком.

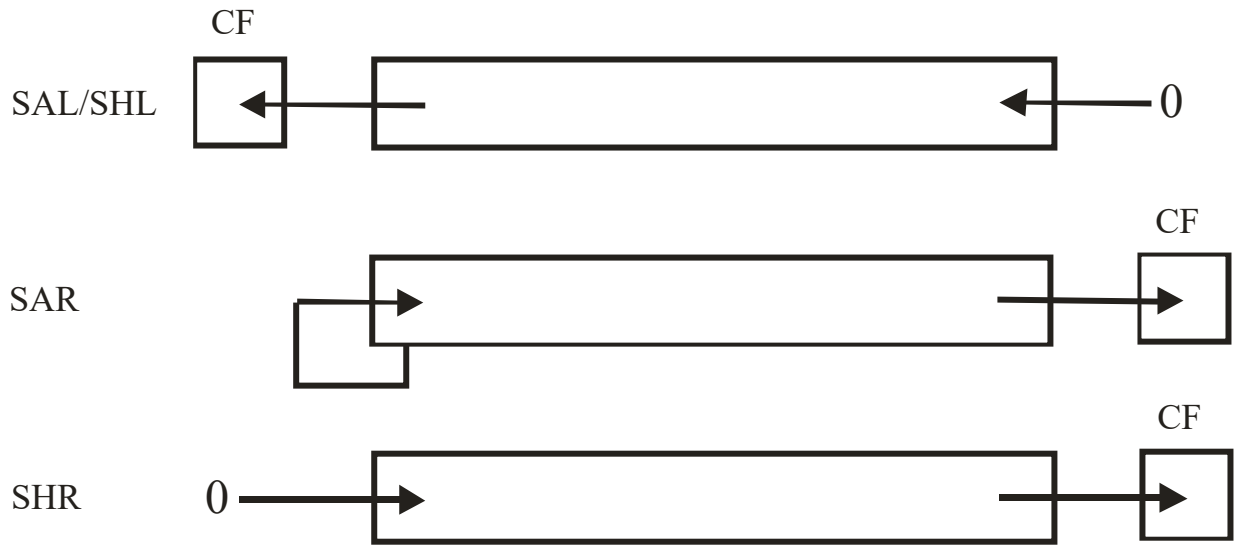

Рисунок 21.

Команды SAL (shift arithmetic left – сдвинуть влево арифметически) и SAR (shift arithmetic right – сдвинуть вправо арифметически) сдвигают числа со знаком.

Команда SAR сохраняет знак операнда, репродуцируя его при выполнении сдвига.

Команда SAL не сохраняет знак, но заносит 1 во флаг переполнения OF в случае изменения знака операнда. При каждом сдвиге операнда команда SAL заносит 0 в вакантный нулевой бит этого операнда.

Команды SHL (shift logical left – сдвинуть влево логически) и SHR (shift logical right – сдвинуть вправо логически) сдвигают числа без знака.

Команда SHL идентична команде SAL.

Команда SHR аналогична команде SHL, но сдвигает операнд не влево, а вправо. При каждом сдвиге операнда команда SHR заносит 0 в вакантный старший бит этого операнда (бит 7 при сдвиге байта, бит 15 при сдвиге слова, бит 31 при сдвиге двойного слова).

Пример. Предположим, что регистр AL содержит 10110100, а флаг переноса CF равен 1.

Команды сдвига воздействуют на регистр AL и флаг CF следующим образом.

После SAL AL, 1: AL= 01101000, CF = 1.
После SAR AL, 1: AL = 11011010, CF = 0.
После SHL AL, 1: AL = 01101000, CF = 1.
После SHR AL, 1: AL = 01011010, CF = 0.

Рассмотрим несколько полезных примеров применения команд сдвига.

Сдвиг операнда на один бит влево удваивает значение операнда (умножает на 2), а сдвиг на один бит вправо уменьшает значение операнда вдвое (делит на 2). Поэтому команды сдвига можно использовать в качестве команд быстрого умножения и деления.

Пример. Умножим и разделим на четыре беззнаковое содержимое регистра AX. При этом регистр CL должен содержать 2.

```
SHL AX, CL    ; Умножить число без знака на 4
SHR AX, CL    ; Разделить число без знака на 4
```

Применение команд сдвига вместо команд умножения и деления позволяет сэкономить время, так как команды сдвига выполняют данные действия в разы быстрее.

Заметим, однако, что программист должен следить за диапазоном как вводимых, так и выводимых данных. Кроме того, операция SAR корректно работает с положительными целыми числами и не всегда корректно – с отрицательными нечетными числами.

В то время как отдельная команда сдвига может умножить или разделить только на степень числа 2, манипулирование несколькими регистрами позволяет выполнить умножение или деление на другие числа.

Пример. Умножим содержимое регистра AX на 10.

```
MOV     BX, AX     ; Сохранить содержимое AX в BX
SHL     AX, 2      ; Сдвинуть AX (умножить на 4)
ADD     AX, BX     ; Сложить с исходным значением AX
                   ; (умножить на 5)
SHL     AX, 1      ; Сдвинуть AX еще раз (умножить на 10)
```

Пример. В системе Windows цвет кодируется при помощи типа данных COLORREF, являющегося четырехбайтным значением. В формате RGB цвет задается через интенсивности трех цветовых компонентов – красного, зеленого и синего. В COLORREF формат RGB представляется следующим образом: в байте 0 задается интенсивность красного (RED) цвета, в байте 1 – зеленого (GREEN), в байте 2 – синего (BLUE), а старший байт – нулевой (см. рисунок 22).

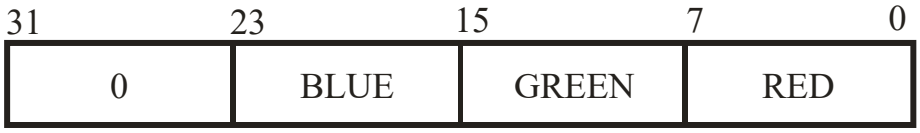

Рисунок 22.

В Windows API имеется макрокоманда RGB, которая позволяет преобразовать интенсивности цветовых компонентов, заданных отдельными числами, в формат COLORREF. Рассмотрим, как такое преобразование сделать на языке ассемблера. Предположим, что каждая из байтовых ячеек RED, GREEN и BLUE содержит интенсивность соответствующего цветового компонента, а результатом является 32-битовая ячейка RGB:

```
XOR     EAX, EAX
OR      AH, BLUE
OR      AL, GREEN
SHL     EAX, 8
OR      AL, RED
```

```
MOV     RGB, EAX
```

Обратное преобразование производится с помощью трех макрокоманд GetRValue, GetGValue, GetBValue, которые выделяют соответствующие цветовые компоненты из значения типа COLORREF. Реализуем такое преобразование на ассемблере:

```
MOV     EAX, RGB
MOV     RED, AL
MOV     GREEN, AH
SHR     EAX, 8
MOV     BLUE, AH
```

8.5.2. Команды циклического сдвига

Команды циклического сдвига, в отличие от команд сдвига, сохраняют сдвинутые за пределы операнда биты, помещая их обратно в операнд (см. рисунок 23).

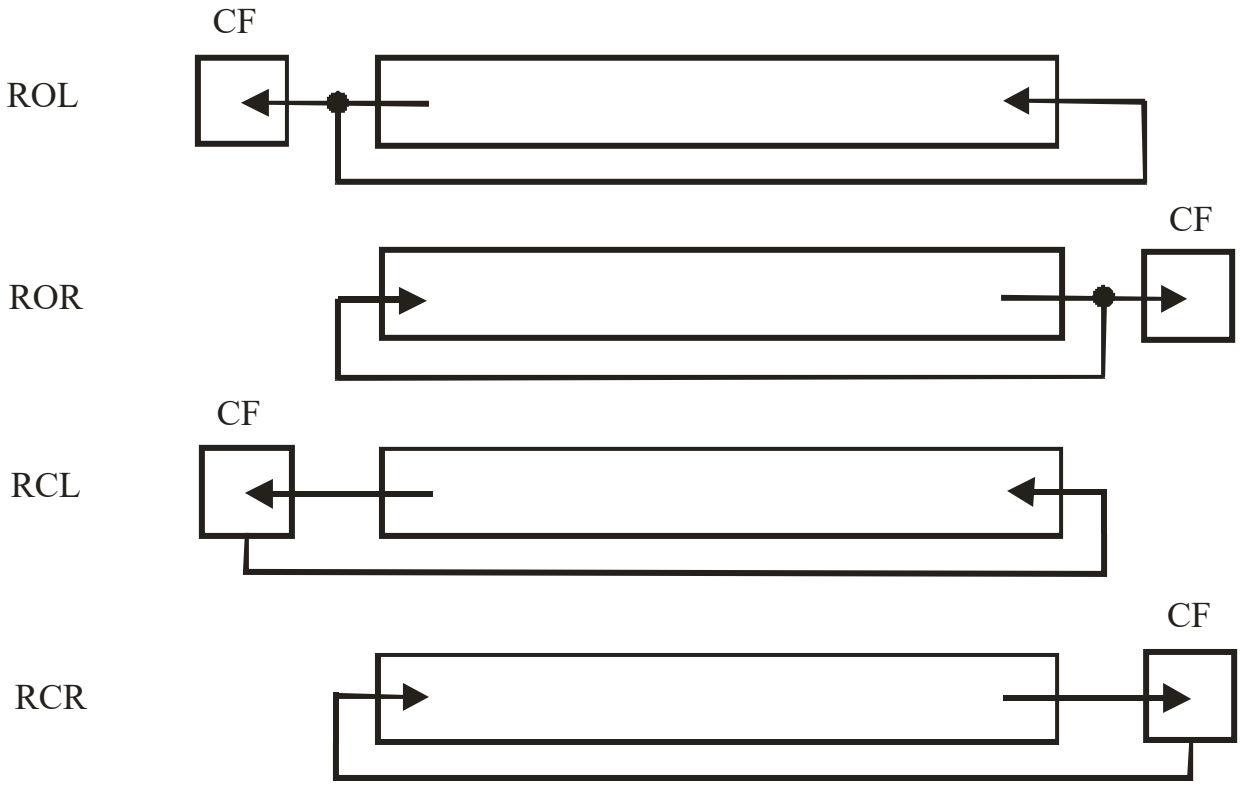

Рисунок 23.

Как и при исполнении команд сдвига, сдвинутый за пределы операнда бит запоминается во флаге переноса CF.

При исполнении команд ROL (rotate left – сдвинуть влево циклически) и ROR (rotate right – сдвинуть вправо циклически) вышедший за пределы операнда бит входит в него с противоположного конца.

При исполнении команд RCL (rotate through carry left – сдвинуть влево циклически вместе с флагом переноса) и RCR (rotate through carry right – сдвинуть вправо циклически вместе с флагом переноса) в противоположный конец операнда помещается значение флага переноса CF.

Все команды циклического сдвига воздействуют только на флаги CF и OF.

Пример. Пусть снова AL = 10110100 и CF = 1. Команды циклического сдвига воздействуют на регистр AL и флаг CF следующим образом.

После ROL AL, l: AL = 01101001, CF = 1.
После ROR AL, 1: AL = 01011010, CF = 0.
После RCL AL, 1: AL = 01101001, CF = 1.
После RCR AL, 1: AL = 11011010, CF = 0.

Команду ROL можно использовать для обмена старшего (биты 4–7) и младшего (биты 0–3) полубайтов числа. Например, в результате циклического сдвига влево числа 26H получим число 62H:

```
MOV     AL, 26H
ROL     AL, 4
```

У микропроцессора есть команды, которые позволяют изменять флаг переноса CF. Команды STC (set carry flag – установить флаг переноса) и CLC (clear carry flag – обнулить флаг переноса) переводят флаг CF в состояния 1 и 0 соответственно. Команда CMC (complement carry flag – обратить флаг переноса) переводит флаг CF в состояние 0, если он имел состояние 1, и наоборот.

Эти команды полезны для установки нужного состояния флага CF перед исполнением команд циклического сдвига с флагом переноса RCL и RCR.

Пример. Рассмотрим, как переписать в регистр BX старшую половину регистра EAX с одновременным ее обнулением в регистре EAX:

```
        MOV     ECX, 16     ; Количество сдвигов для EAX
ML:
        CLC                 ; Сброс флага CF в 0
        RCL     EAX, 1      ; Сдвиг крайнего левого бита из EAX в CF
        RCL     BX, 1       ; Перемещение бита из CF справа в BX
        LOOP    ML          ; Цикл 16 раз
        ROL     EAX, 16     ; Восстановить правую часть EAX
```

8.5.3. Команды сдвига двойной точности

В отличие от рассмотренных выше команд сдвига, у этих команд не два, а три операнда.

Команда SHLD (shift left double – сдвиг влево удвоенный) имеет следующий формат:

SHLD приемник, источник, счетчик

Команда выполняет логический сдвиг влево операнда-приемника на количество разрядов, указанных в счетчике (см. рисунок 24). Освободившиеся в результате сдвига разряды приемника заполняются старшими битами операнда-источника. При этом значение источника не изменяется, но меняется ряд флагов (в том числе SF, ZF, PF, CF).

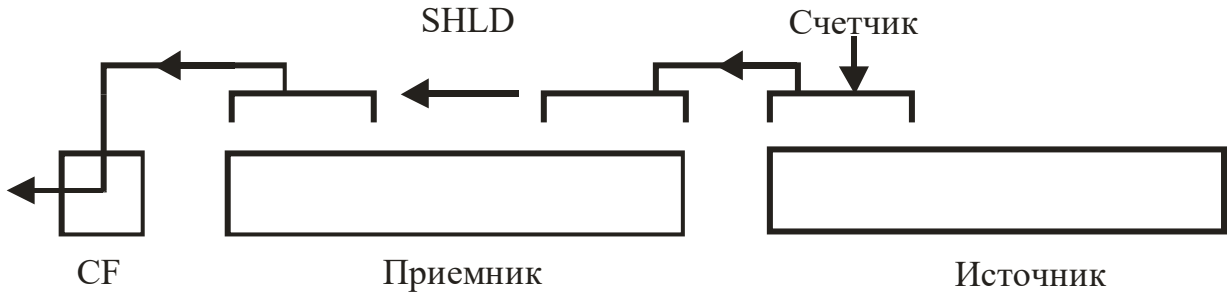

Рисунок 24.

Команда SHRD (shift right double – сдвиг вправо удвоенный) имеет такой же формат и выполняет логический сдвиг вправо операнда-приемника на количество разрядов, указанных в третьем операнде (см. рисунок 25). Освободившиеся в результате сдвига разряды приемника заполняются младшими битами операнда-источника.

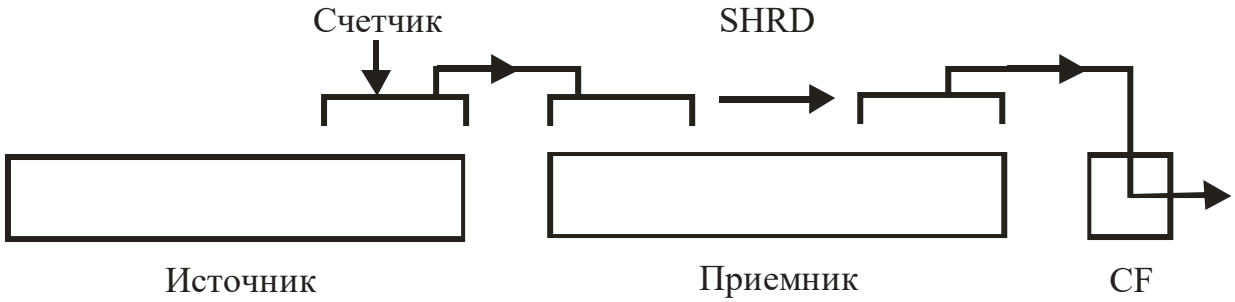

Рисунок 25.

Приемник может располагаться либо в памяти, либо в регистре. Источник может находиться только в регистре. В качестве счетчика может быть задан либо регистр CL, либо 8-разрядная константа.

Команды осуществляют сдвиги на величину до 32 разрядов (значение счетчика сдвигов может лежать в диапазоне 0...31), но за счет особенностей задания операндов и алгоритма работы эти команды можно использовать для работы с полями длиной до 64 битов.

Пример. Рассмотрим, как можно осуществить сдвиг влево на 16 битов поля из 64 битов.

```
.DATA
POLE_L    DD    0B21187F5H        ; Младшая часть
POLE_H    DD    45FF6711H         ; Старшая часть
.CODE
MOV       CL, 16                  ; Загрузка счетчика сдвигов в CL
MOV       EAX, POLE_L
SHLDPOLE_H, EAX, CL       ; POLE_H=6711B211H
SHL       POLE_L, CL              ; POLE_L=87F50000H
```

9. Двоично-десятичные числа

Микропроцессор может выполнять арифметические команды не только над двоичными числами, но также над десятичными числами без знака, имеющими специальное двоично-десятичное представление и часто называемыми двоично-десятичными числами (BCD-числами – Binary Coded Decimal).

9.1. Форматы хранения двоично-десятичных чисел

Микропроцессор хранит двоично-десятичные числа в виде последовательностей байтов без знака в упакованном или неупакованном формате.

Каждый байт *упакованного* двоично-десятичного числа содержит две двоично-десятичные цифры. При этом двоичный код старшей цифры числа занимает четыре старших бита (старшую тетраду). Следовательно, один упакованный десятичный байт может содержать значения от 00 до 99.

Пример. Число 23 имеет следующее представление в упакованном формате: 0010 0011.

Каждый байт *неупакованного* двоично-десятичного числа содержит только одну двоично-десятичную цифру в четырех младших битах (в младшей тетраде). Следовательно, один неупакованный десятичный байт может содержать лишь значение от 0 до 9.

Пример. Число 23 имеет следующее представление в неупакованном формате: 0000 0010 0000 0011.

Если необходимо произвести преобразование двух неупакованных BCD-чисел в упакованное BCD-число, то для этого удобно воспользоваться командами логического сдвига влево SHL и командой логического ИЛИ. Пусть старшая цифра находится в регистре BL, а младшая – в регистре AL, результат помещается в регистр AL. Получаем

```
SHL   BL, 4       ; Сдвинуть старшую цифру в старшие четыре бита BL
OR    AL, BL      ; Получить упакованное число слиянием AL и BL
```

Двоично-десятичные числа нужны в деловых приложениях, где числа должны быть большими и точными.

Микропроцессор при арифметических операциях трактует все операнды только как двоичные числа.

Это хорошо в том случае, когда операнды действительно являются двоичными числами. Если же они являются двоично-десятичными, то результаты будут ошибочными. Для компенсации таких ошибок имеется группа команд коррекции, которые обеспечивают получение правильного результата после выполнения операций над двоично-десятичными числами. Кроме того, в регистре FLAGS имеется вспомогательный флаг переноса AF (auxiliary carry flag), который специально применяется для команд, работающих с двоично-десятичными числами.

Под каждое двоично-десятичное число в памяти отводят группу соседних байтов – столько, сколько нужно. Порядок, в котором цифры числа занимают эти байты, вообще говоря, не фиксируется и определяется программистом. Дело в

том, что обработка этих чисел ведется по цифрам и переход от цифры к цифре организует сам программист, а он может с одинаковым успехом переходить от очередного байта как к следующему байту, так и к предыдущему. Но для определенности мы в дальнейшем будем располагать левые (старшие) цифры числа в байтах с меньшими адресами, а правые цифры – с большими.

Ниже приведены представления числа 592 в упакованном (рисунок 26) и неупакованном (рисунок 27) форматах, A – адрес первого из байтов, занятых числом.

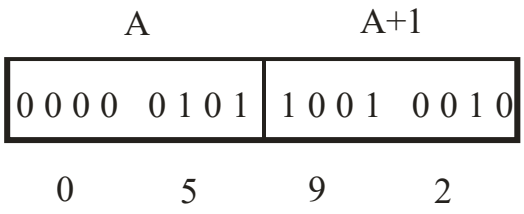

Рисунок 26.

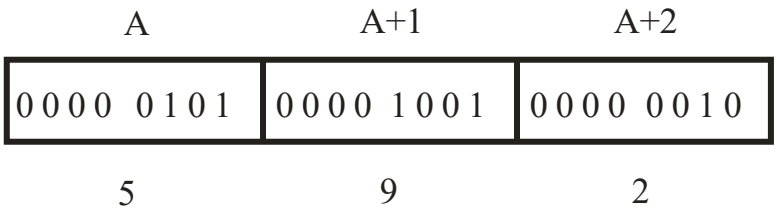

Рисунок 27.

В языке ассемблера переменные, значения которых трактуются как двоично-десятичные числа, можно описать директивой DB с несколькими операндами:

```
PACK    DB    5H, 92H      ; Число 592 в упакованном виде
UNPACK  DB    5, 9, 2      ; Число 592 в неупакованном виде
```

Обратите внимание, что для упакованного числа в одном операнде надо указывать две цифры, причем указывать их в шестнадцатеричной системе, чтобы каждая из цифр при трансляции заменялась на соответствующую четверку битов независимо от другой цифры (если в качестве операнда указать десятичное число 92, тогда оно заменится на 0101100B, а не на 10010010B).

9.2. Коррекция результата сложения

Коррекция результата сложения двоично-десятичных чисел осуществляется командами AAA (для неупакованного формата) и DAA (для упакованного формата). В них не требуется наличия операнда: предполагается, что корректируемое значение находится в регистре AL.

Команда AAA преобразует содержимое регистра AL в правильную *неупакованную* двоично-десятичную цифру в младших четырех битах регистра AL (и заполняет нулями старшие четыре бита).

Она используется в следующем контексте:

```
ADD   AL, BL  ; Сложить неупакованные числа, находящиеся в AL и
AAA           ; BL и преобразовать результат в неупакованное число
```

Если результат превышает 9, то команда AAA добавляет 1 к содержимому регистра AH (чтобы учесть избыточную цифру). Команда изменяет ряд флагов.

Пример. Рассмотрим сложение неупакованных десятичных цифр 6 и 3, ответ: неупакованная десятичная цифра 9 (см. рисунок 28). В данном случае коррекция не нужна.

Рисунок 28.

	0	0	0	0	0	1	1	0
+	0	0	0	0	0	0	1	1
=	0	0	0	0	1	0	0	1

Рассмотрим сложение неупакованных десятичных цифр 6 и 7 (см. рисунок 29).

	0	0	0	0	0	1	1	0
+	0	0	0	0	0	1	1	1
=	0	0	0	0	1	0	1	1

Рисунок 29.

Результат не является неупакованной десятичной цифрой. Правильный результат представляет собой две неупакованные десятичные цифры (1 и 3): 0000 0001 и 0000 0011. Для коррекции необходимо от полученного результата вычесть 10. Но в микропроцессоре корректировка производится чуть иначе. Прибавим к младшей тетраде результата число 6 (см. рисунок 30)

		1	1	0	1
+		0	1	1	0
=	1	0	0	1	1

Рисунок 30.

В результате младшая тетрада содержит младшую цифру правильного ответа. Именно таким образом и происходит корректировка.

Команда DAA преобразует содержимое регистра AL в две правильные *упакованные* двоично-десятичные цифры. Она используется в следующем контексте:

```
ADD     AL, BL    ; Сложить упакованные числа AL и BL
DAA               ; и преобразовать результат в упакованное число
```

Если результат превышает 99 (предельное значение для упакованных чисел), то команда DAA добавляет 1 к содержимому регистра AH. Команда изменяет ряд флагов.

9.3. Коррекция результата вычитания

Коррекция результата вычитания двух двоично-десятичных чисел осуществляется командами AAS (для неупакованного формата) и DAS (для упакованного формата). При их исполнении предполагается, что корректируемое число находится в регистре AL.

Команда AAS преобразует содержимое регистра AL в правильную *неупакованную* двоично-десятичную цифру в младших четырех битах регистра AL (и обнуляет старшие четыре бита).

Она используется в следующем контексте:

```
SUB     AL, BL    ; Вычесть неупакованное число (содержимое BL) из AL
AAS               ; и преобразовать результат в неупакованное число
```

Если результат превышает 9, то команда AAS вычитает 1 из содержимого регистра AH. Команда изменяет ряд флагов.

Принцип корректировки, является обратным тому, который используется при сложении: он основан на вычитании числа 6 из младшей тетрады.

Команда DAS преобразует содержимое регистра AL в две правильные упакованные десятичные цифры.

Она используется в следующем контексте:

```
SUB     AL, BL  ; Вычесть упакованное число (содержимое BL)
DAS             ; из AL и преобразовать результат в упакованное число
```

Если результат превышает 99 (предельное значение для упакованных чисел), то команда DAS вычитает 1 из содержимого регистра AH. Команда изменяет ряд флагов.

9.4. Коррекция результата умножения

Команда AAM преобразует результат предшествующего умножения байтов в два правильных неупакованных операнда. Она считает, что произведение двойного размера находится в регистрах AH и AL, и возвращает неупакованные операнды в регистрах AH и AL.

Чтобы команда AAM работала правильно, исходные множимое и множитель должны быть правильными неупакованными байтами.

Для выполнения преобразования команда AAM делит значение регистра AL на 10 и запоминает частное и остаток в регистрах AH и AL соответственно. Команда изменяет ряд флагов.

У микропроцессора нет команды умножения упакованных десятичных чисел. Для выполнения этой операции необходимо распаковать эти числа, перемножить их, воспользоваться командой AAM, а затем упаковать результат.

9.5. Коррекция результата деления

Все ранее описанные команды десятичной коррекции (AAA, DAA, AAS, DAS и AAM) выполняли действия над результатом операции. В противоположность им команда AAD должна исполняться непосредственно перед операцией деления.

Команда AAD преобразует неупакованное делимое в двоичное значение и загружает его в регистр AL. Для этого она умножает старшую цифру делимого (содержимое регистра AH) на 10 и добавляет полученный результат к младшей цифре, находящейся в регистре AL. Затем она обнуляет содержимое регистра AH.

Приведем типичный пример применения команды AAD:

```
AAD          ; Скорректировать неупакованное делимое в AH:AL,
DIV  BL      ; а затем выполнить деление
```

10. Контрольные вопросы

1. Опишите особенности языка ассемблера.
2. Дайте характеристику разным группам регистров микропроцессора.
3. С какой целью используются директивы SEGMENT, ASSUME, END?
4. Как использовать упрощенные директивы сегментации?
5. Какую роль играют директивы определения данных?
6. Как выполнить пересылку данных?
7. Чем отличаются команды ADD, ADC, XADD, INC?
8. Чем отличаются команды SUB, SBB, DEC, NEG?
9. Как выполнить умножение и деление?
10. В каких случаях полезны команды преобразования типа?
11. Чем отличаются регистровая, непосредственная и прямая адресация?
12. Чем отличаются косвенная регистровая адресация и адресация по базе?
13. Для каких целей удобно использовать прямую адресацию с индексированием и адресацию по базе с индексированием?
14. Как описать процедуру?
15. Как изменяется содержимое регистра EIP при выполнении команд CALL, RET, JMP?
16. Как совместно использовать команды условной передачи управления и команду CMP?
17. Для чего нужны команды SETx, CMOVx?
18. Как выполняются команды управления циклами?

19. Опишите основы взаимодействия языков C++ и ассемблера, передачу управления в подпрограмму и обратно.
20. Как организовать переду данных из программы на языке C++ в подпрограмму на языке ассемблера через глобальные переменные?
21. Как организовать переду данных из программы на языке C++ в подпрограмму на языке ассемблера через аргументы функции?
22. Как возвратить данные из подпрограммы на языке ассемблера в программу на языке C++?
23. Как вызвать подпрограмму на языке C++ из программы на языке ассемблера?
24. Как устроен стековый фрейм?
25. Как использовать библиотечных функций языка Си в программах или подпрограммах на языке ассемблера?
26. Каковы особенности использования вставок на языке ассемблера в программе на языке C++?
27. Как выполняются команды AND, OR, XOR, NOT, TEST?
28. Что делают команды сканирования битов, команды проверки и модификации битов?
29. Как выполняются команды сдвига?
30. Как выполняются команды циклического сдвига?
31. Как выполняются команды сдвига двойной точности?
32. Каковы форматы хранения двоично-десятичных чисел?
33. С какой целью производится коррекция результатов арифметических команд для двоично-десятичных чисел?

Список литературы

1. Абель П. Язык Ассемблера для IBM PC и программирование. – Высшая школа, 1992.
2. Галисеев Г.В. Ассемблер для Win 32. Самоучитель. – М.: Издательский дом "Вильямс", 2007.
3. Голубь Н.Г. Искусство программирования на Ассемблере: лекции и упражнения. – СПб.: ДиаСофтЮП, 2002.
4. Зубков С.В. Assembler для DOS, Windows и Unix. – М.: ДМК Пресс, 2008.
5. Ирвин К. Язык ассемблера для процессоров Intel. – М.: Издательский дом "Вильямс", 2005.
6. Крупник А.Б. Изучаем Ассемблер. – СПб.: Питер, 2005.
7. Крупник А. Ассемблер. Самоучитель. – СПб.: Питер, 2005.
8. Магда Ю.С. Ассемблер для процессоров Intel Pentium. – СПб.: Питер, 2006.
9. Пильщиков В.Н. Программирование на языке ассемблера IBM PC. – М.: ДИАЛОГ-МИФИ, 1999.
10. Пирогов В.Ю. Ассемблер для Windows. – СПб.: БХВ-Петербург, 2008.
11. Пирогов В.Ю. Ассемблер на примерах. – СПб.: БХВ-Петербург, 2005.

12. Рудаков П.И., Финогенов К.Г. Язык ассемблера: уроки программирования. – М.: ДИАЛОГ-МИФИ, 2001.

13. Скэнлон Л. Персональные ЭВМ IBM PC и XT. Программирование на языке ассемблера. – М.: Радио и связь, 1989.

14. Трой Д. Программирование на языке Си для персонального компьютера IBM PC. – М.: Радио и связь, 1991.

15. Юров В.И. Assembler. Учебник для вузов. – СПб.: Питер, 2006.

16. Юров В.И. Assembler. Практикум. – СПб.: Питер, 2006.

Содержание