

Глава 6. Язык Ассемблера

Отказ от языка ассемблера был яблоком раздора в наших садах Эдема. Языки, использование которых приводит к растранижению машинного времени, греховны.

Алан Перлис

Программирование в первую очередь – искусство, превращающееся в науку при непрерывной практике, а Ассемблер – единственно верный путь, воплощающий науку в искусство.

Edmond / HI-TECH

6.1. Понятие о языке Ассемблера

Многие вещи нам непонятны не потому, что наши понятия слабы; но потому, что сии вещи не входят в круг наших понятий.

Козьма Прутков

То, что не удаётся запрограммировать на Ассемблере, приходится паять.

Наличие большого количества форматов данных и команд в архитектурах некоторых современных ЭВМ приводит к дополнительным существенным трудностям при программировании на машинном языке. Для упрощения процесса написания программ для ЭВМ был разработан язык-посредник, названный *Ассемблером*, который, с одной стороны, должен допускать написание любых машинных команд,¹ а с другой стороны – позволять автоматизировать и упростить процесс составления программ в машинном коде. Как уже говорилось ранее, предшественниками Ассемблеров были так называемые языки псевдокодов, такой язык псевдокода использовался в этой книге при изучении архитектур учебных машин.

Для перевода с языка Ассемблера на язык машины² используется специальная программа-переводчик (транслятор), также называемая *Ассемблером* (от английского слова "assembler" – "сборщик"). В зависимости от контекста, если это не будет вызывать неоднозначности, под словом "Ассемблер" будет пониматься или сам язык программирования, или транслятор с этого языка.

В этой книге не будут рассматриваться все возможности языка Ассемблера, так как для целей изучения архитектуры ЭВМ понадобится только некоторое подмножество этого языка, только оно и будет использоваться в примерах этой книги.

Рассмотрим, что, например, должен делать компилятор при переводе с языка Ассемблера на язык машины:

- заменять мнемонические обозначения кодов операций на соответствующие машинные коды операций (например, для нашей учебной машины УМ-3, ВЧЦ → 12);
- автоматически распределять память под хранение переменных, что позволяет программисту не заботиться о конкретном адресе переменной, если ему всё равно, где она будет расположена;
- подставлять в программе вместо имён переменных их значения, обычно значение имени переменной – это смещение этой переменной от начала памяти (или некоторого сегмента);
- преобразовывать числа, написанные в программе в различных системах счисления во внутреннее машинное представление (в машинную систему счисления).

¹ Впрочем, для некоторых машинных команд в языке Ассемблера не предусмотрено соответствующих им мнемонических обозначений кодов операций. Обычно это такие команды, которые программисту не рекомендуется вставлять в программу в явном виде, эту работу следует поручить компилятору с языка Ассемблер. Например, это уже упоминавшаяся ранее команда-префикс смены длины регистра с кодом операции bbl.

² Как будет показано позже, на самом деле чаще всего производится перевод не на язык машины, а на специальный промежуточный язык, который называется *объектным* языком.

В конкретном Ассемблере обычно существует много полезных возможностей для более удобного написания программ, что возлагает на Ассемблер дополнительные функции, однако при этом должны выполняться следующие требования (они вытекают из принципов Фон Неймана, если, конечно, они выполняются в конкретном компьютере):

- возможность помещать в любое определённое программистом место основной памяти любую команду или любые данные (однако, как уже говорилось ранее, необходимо учитывать, что некоторые области оперативной памяти могут быть закрыты на запись);
- возможность выполнять любые данные как команды и работать с командами, как с данными (например, складывать команды как числа);
- возможность выполнить любую команду из языка машины.¹

6.2. Применение языка Ассемблера

Цель программирования – не создание программ, а получение результатов вычислений.

Д. Ван Тассел

Общеизвестно, что программировать на Ассемблере трудно. Как Вы знаете, сейчас существует много различных языков высокого уровня, которые позволяют затрачивать намного меньше усилий при написании программ. Так, считается, что на один оператор языка высокого уровня, при программировании его на Ассемблере, приходится тратить примерно 10 предложений. Кроме того, понимать, отлаживать и модифицировать программы на Ассемблере значительно труднее, чем программы на языках высокого уровня. На рис. 6.1 показана взаимосвязь языков программирования высокого уровня (их ещё называют машинно-независимыми), языков низкого уровня (машинно-ориентированных) и собственно языка машины.

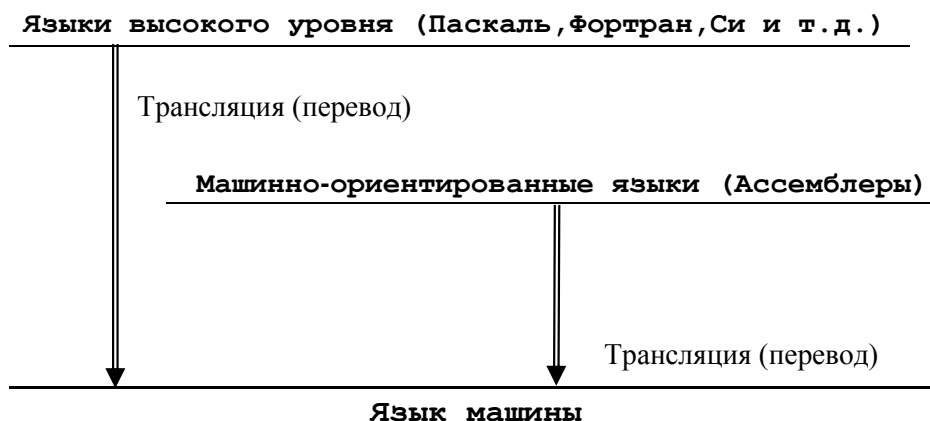


Рис. 6.1. Взаимосвязь языков программирования разных уровней.

Естественно, возникает вопрос, когда у программиста может появиться необходимость при написании своих программ использовать не более удобный язык программирования высокого уровня, а перейти на язык низкого уровня (Ассемблер). В настоящее время можно указать две области, в которых использование языка Ассемблера оправдано, а зачастую и необходимо.

Во-первых, это так называемые машинно-зависимые системные программы, обычно они управляют различными устройствами компьютера (такие программы, как правило, называются драйверами). В этих системных программах используются специальные машинные команды, которые нет необходимости применять в обычных (или, как говорят, прикладных) программах. Эти команды невозможно или весьма затруднительно задать в программе на языке высокого уровня. Кроме того, обычно от драйверов требуется, чтобы они были компактными и выполняли свою работу за минимально возможное время.

¹ Современные ЭВМ могут работать в так называемом привилегированном (или защищённом) режиме [16]. В этом режиме программы обычных пользователей, не имеющие соответствующих привилегий, не могут выполнять некоторое подмножество особых (привилегированных) команд из языка машины. Аналогично на современных ЭВМ существует защита памяти, при этом одна программа не может иметь доступ (писать и читать) в память другой программы. Привилегированный режим работы и защита памяти будет изучаться в главе, посвящённой мультипрограммному режиму работы ЭВМ.

Вторая область применения Ассемблера связана с оптимизацией выполнения *больших* программ,¹ которые требуют много времени для счёта. Часто программы-переводчики (трансляторы) с языков высокого уровня дают не совсем эффективную программу на машинном языке. Причина этого заключается в том, что такие программы могут иметь специфические особенности, которые не сможет учесть транслятор. Особенно это касается программ вычислительного характера, в которых большую часть времени выполняется очень небольшой по длине (около 1-3%) участок программы (обычно называемый главным циклом). Для повышения эффективности выполнения этих программ могут использоваться так называемые многоязыковые системы программирования, которые позволяют записывать части программы на разных языках. Обычно основная часть оптимизируемой программы записывается на языке программирования высокого уровня (Фортране, Паскале, Си и т.д.), а критические по времени выполнения участки программы – на Ассемблере. Скорость работы всей программы при этом может значительно увеличиться. Заметим, что часто это единственный способ заставить сложную программу дать результат за приемлемое время.

Итак, область применения языка Ассемблер в программировании непрерывно сокращается. В то же время, хорошему программисту совершенно необходимо ясно представлять, как написанные им конструкции на языках высокого уровня будут преобразовываться соответствующими трансляторами в машинный код. Умея мыслить в терминах языка низкого уровня, программист будет более ясно понимать, что происходит при выполнении его программы на ЭВМ, и как с учётом этого разрабатывать программы на языках высокого уровня.

6.3. Структура программы на Ассемблере

Язык программирования имеет низкий уровень, если в программах приходится уделять внимание несущественному.

Алан Перлис

При дальнейшем изучении архитектуры компьютера нам придётся писать как фрагменты, так и полные программы на языке Ассемблер. Для написания этих программ будет использоваться одна из версий языка Ассемблер (и соответствующего компилятора), так называемый Макроассемблер версии 6.14 (MASM-6.14).

Достаточно полное описание этого языка приведено в учебниках [5-7], их изучение является желательным для хорошего понимания материала. В этой книге будут подробно изучаться только те особенности и тонкие свойства языка Ассемблера, которые необходимы для хорошего понимания архитектуры изучаемой ЭВМ и написания простых полных (с вводом и выводом) программ.

Изучение языка Ассемблера начнём с рассмотрения общей структуры программы на этом языке. Полная программа на языке Ассемблера состоит из одного или более *модулей*. Таким образом, Ассемблер принадлежит к классу так называемых модульных языков. В таких языках вся программа может разрабатываться, писаться и отлаживаться как набор относительно независимых друг от друга программных частей – модулей.² В каком смысле модуль является *независимой* единицей языка Ассемблер, будет объяснено несколько позже, когда будет изучаться тема "Модульное программирование". Наши первые программы будут содержать всего один модуль,³ но позже будут рассмотрены и простые многомодульные программы.

Каждый модуль обычно содержит описание одной или нескольких *секций* памяти. В соответствии с принципом Фон Неймана, программист имеет право размещать в любой секции как числа, так и команды. Такой подход, однако, ведёт к плохому стилю программирования, программа перестаёт легко читаться и пониматься, её труднее отлаживать и модифицировать. Программирование становится более лёгким и надёжным, если размещать команды программы в одной секции, а данные – в других. Весьма редко программисту будет выгодно размещать, например, данные среди команд или

¹ Обычно в программистской литературе *большими* принято называть программы, содержащие порядка 100 тысяч и более строк исходного текста. Кроме того, большими называются и программы, требующие много (сотни и тысячи) часов машинного времени для своего выполнения, а также иногда требуют и завершения счёта к определённому времени (например, прогноз погоды).

² Иногда для такой функционально-независимой части программы употребляются и другие термины, имеющие похожий смысл, например, "пакет", "библиотека стандартных программ", "библиотека классов" и др.

³ Точнее, один (головной) модуль мы будем писать сами, а остальные необходимые (библиотечные) модули будут (автоматически) подключаться к нашей программе в случае необходимости.

разместить команду среди данных. Другими словами, можно отступить от принципа Фон Неймана, и для обеспечения надёжности секции кода аппаратно закрыть на запись, а секции данных закрыть на выполнение из них команд.

Итак, модуль в основном состоит из описаний секций, в секциях находятся все команды и переменные. Вне секций могут располагаться только так называемые *директивы* языка Ассемблер, о которых будет сказано немного ниже. Пока лишь отметим, что чаще всего директивы не определяют в программе ни команд, ни переменных (именно поэтому они и могут стоять *вне секций*).¹

Описание каждой секции, в свою очередь, состоит из *предложений* (statements) языка Ассемблера. Каждое предложение языка Ассемблера занимает отдельную строчку программы, исключение из этого правила будет отмечено особо. Далее рассмотрим различные классы предложений Ассемблера.

6.4. Классификация предложений языка Ассемблер

Язык, который не меняет Вашего представления о программировании, достоин изучения.

Алан Перлис

Классификация предложений Ассемблера проводится по тем функциям, которые они выполняют в программе. Заметим, что эта классификация иногда немного отличается от той, которая приведена в рекомендованных учебниках [5-7].

- **Комментарии.** Различают однострочные и многострочные комментарии. Однострочный комментарий начинается с символа ";", перед которым в строке могут находиться только пробелы и знаки табуляции, например:

; это строка-комментарий

Многострочный комментарий может занимать несколько строк текста программы. Будем для унификации терминов считать её неким частным типом предложения, хотя не все авторы учебников по Ассемблеру придерживаются этой точки зрения. Способ записи этих комментариев:

```
comment <символ-ограничитель> <комментарий>
< строки – комментарии >
<комментарий> <символ-ограничитель> <комментарий>
```

здесь служебное имя **comment** определяет многострочный комментарий, а символ-ограничитель задаёт его границы, он отдалённо эквивалентен символам начала и конца комментария { и } в языке Паскаль, этот символ не должен встречаться внутри самого комментария, например:

```
comment @ Вот что делает программа:
< строки – комментарии без @>
Последняя строка комментария @ Надеюсь, понятно ?
```

- **Команды.** Почти каждому предложению языка Ассемблера этого типа будет соответствовать одной команде (instruction) на языке машины. Иногда получаются несколько "тесно связанных" команд, обычно говорят, что перед основной командой ставятся одна или несколько так называемых команд-префиксов.

- **Резервирование памяти.** В той секции, где они записаны, резервируются области памяти для хранения переменных. Это некоторый аналог описания переменных **var** в языке Паскаль. Заметим, что во многих учебниках такие предложения называют *директивами* резервирования памяти. Полные правила записи этих предложений можно посмотреть в учебниках [5-7], здесь приведены лишь некоторые примеры с комментариями.

Предложение	Количество памяти
A db ?	1 байт
B dw 1	2 байта (слово)
C dd ?	4 байта (двойное слово)
D df 0	6 байт
E dq ?	8 байт

¹ Исключением, например, является директива **include**, на место которой подставляется некоторый текстовый файл. Этот файл может содержать описание одной или нескольких секций или же наборы фрагментов программ (так называемые макроопределения). С примером использования этой директивы Вы вскоре познакомитесь.

F dt ?	10 байт
---------------	---------

В этих примерах описаны переменные разной длины.¹ Переменные с именами В и D имеют начальные значения, а остальные переменные, как в стандарте языка Паскаль, не будут иметь конкретных начальных значений, что отмечено символом вопросительного знака в поле параметров. Однако по принципу Фон Неймана ничто не мешает нам работать напрямую с одним или несколькими байтами, расположенными в любом месте памяти. Например, команда

```
mov ax, B+1
```

будет читать на регистр AX слово, *второй* байт которого располагается в конце переменной В, а *первый* – в начале переменной С (надо помнить о "перевернутом" хранении слов в памяти!). Поэтому следует быть осторожными и не считать А, В, С и D отдельными, "независимыми" переменными в смысле языка Паскаль, это просто именованные области памяти. Разумеется, в понятно написанной программе эти области лучше использовать так, как они описаны, то есть с помощью присвоенных им имён.

В качестве ещё одного примера резервирования памяти рассмотрим предложение

```
D dd 20 dup (?)
```

Оно резервирует в секции 20 подряд расположенных двойных слов с неопределёнными начальными значениями. Это можно назвать резервированием памяти под массив из 20 элементов, но, при этом в Ассемблере также не потеряна возможности работать с произвольными байтами и словами из области памяти, зарезервированной под такой массив.

- **Директивы.** В некоторых учебниках директивы называют **командами Ассемблера**, что хорошо отражает их назначение в программе. Эти предложения, как уже упоминалось, за редким исключением не порождают в машинной программе никакого кода, т.е. команд или переменных. Директивы используются программистом для того, чтобы давать компилятору Ассемблера определённые указания, задавать синтаксические конструкции и управлять работой Ассемблера при компиляции (переводе) программы на язык машины.

Частным случаем директивы будем считать и предложение-метку, которая приписывает имя (метку) непосредственно следующему за ней предложению Ассемблера. Так, в приведённом ниже примере метка `Next_Statement_Name` является именем следующего за ней предложения, таким образом, у этого предложения будет две метки (два имени):

```
Next_Statement_Name:
L: mov eax, 2
```

- **Макрокоманды.** Этот класс предложений Ассемблера относится к *макросредствам* языка, и будет подробно изучаться далее в этой книге. Пока надо лишь сказать, что на место макрокоманды при трансляции в программу по определённым правилам подставляется некоторый набор (возможно и пустой) предложений Ассемблера.

Теперь рассмотрим структуру одного предложения. За редкими исключениями, каждое предложение может содержать от одного до четырёх *полей*: поле *метки*, поле *кода операции*, поле *операндов* и поле *комментария* (как обычно, квадратные скобки в описании синтаксиса указывают на необязательность заключённой в них конструкции):

```
[<метка>[:]] КОП [<операнды>] [; комментарий]
```

Как видно, все поля предложения, кроме кода операции, являются необязательными и могут отсутствовать в конкретном предложении. Метка является именем предложения, как и в Паскале, имя обязано начинаться с буквы, за которой следуют только буквы (а также приравненные к буквам символы) и цифры. В отличие от языка Free Pascal, однако, в языке Ассемблера кроме знака подчёркивания к буквам относятся также символы '\$', '@', '?' и даже точка (правда, только в первой позиции имени). Как и в Паскале, длина имени ограничена максимальной длиной строки предложения Ассемблера. Если после метки стоит двоеточие, то это указание на то, что данное предложение может рассматриваться как *команда*, т.е. на него Ассемблеру можно, как говорят, *передать управ-*

¹ В качестве служебного слова **db** можно использовать синонимы **byte** и **sbyte** (они несколько различаются по семантике использования), вместо **dw** можно использовать **word** и **sword**, вместо **dd** можно использовать синонимы **dword** и **sdword**, а вместо **dt** синоним **tbyte**. В некотором смысле этим синонимам в языке Free Pascal можно указать такие соответствующие типы:

```
byte  -> byte      sbyte -> shortint  word   -> word
sword -> integer  dword -> longword  sdword -> longint
```

ление. Как будет показано позже при изучении команд переходов, наличие у метки двоеточия для Ассемблера есть признак того, что надо использовать прямой, а не косвенный тип перехода по этой метке.

Операнды, если их в предложении несколько, отделяются друг от друга запятыми. В простейших случаях в качестве операндов используются имена меток и переменных, а также константы. В более сложных случаях операндами являются так называемые адресные выражения, которые будут рассмотрены позже.

В очень редких случаях предложения языка Ассемблера имеют другую структуру, например, *директива* присваивания значения числовой макропеременной (с этими переменными Вы будете знакомиться при изучении макросредств языка):

```
K = K+1
```

Длинные предложения Ассемблера можно переносить на следующую строку, указывая в конце предыдущей символ \ (обратный слэш),¹ например:

```
My_Long_Label_Name: mov eax, \
                      My_Longint_Variable \
                      [eax+8*ebx] ;           предложение в три строки
```

Отметим, что *комментарий* и строки в кавычках и апострофах переносить таким образом на новую строку нельзя (как и в тексте программы на Паскале, перенос на новую строку можно делать только *между* лексемами). Общая длина всех таких строк не должна превышать 512 символов. После символа \ можно задавать дополнительный комментарий, например:

```
mov eax,           \ Первый комментарий
[eax+8*ebx]       \ Второй комментарий
; Третий комментарий
```

Как и в большинстве языков программирования, в Ассемблере есть набор служебных (зарезервированных, ключевых) слов, которые можно употреблять в тексте программы только в том смысле, который предписывается языком программирования. В основном это коды операций (**add**, **sub**, **inc** и т.д.), директивы (**proc**, **macro**, **end**, **.code** и т.д.). К сожалению, в набор служебных входят и некоторые "неожиданные" имена, например **C** (означает, конечно же, язык C, точнее его соглашения о связях, о чём будет говориться далее), **Pascal**, **Name** и другие.

Как и в Паскале, в языке Ассемблера принято такое же соглашение по семантике всех *служебных* имён: большие и маленькие буквы не различаются. Таким образом, имена можно писать как заглавными, так и прописными буквами, например, имена **EAX**, **EAX**, **eaX** и **eax** обозначают в Ассемблере *один и тот же* регистр. Для имён пользователя, наоборот, установлена такая опция работы компилятора, что эти имена чувствительны к регистру символов, так что **Name** и **name** – это *разные* имена. По умолчанию с точки могут начинаться только служебные имена.²

Итак, например, имя кода операции **sub** нельзя использовать в качестве, скажем имени переменной или процедуры. Однако, как это часто бывает, когда чего-то "нельзя, но оч-е-нь хочется", Ассемблер допускает исключить некоторые слова из списка ключевых, это делается с помощью опции компилятора **option NOKEYWORD**, например:

```
div eax
option NOKEYWORD:<div inc>
div dd ?
inc eax; ОШИБКА, не описано имя inc
```

Можно рекомендовать исключить из служебных такие часто используемые имена, как **C** и **Name**:

```
option NOKEYWORD <C Name>
```

Данная опция действует от места её объявления вниз по тексту программы. Разумеется, исключения из списка ключевых таких имён, как **.code** или, тем более **end**, приводит к ошибкам компиляции "чуть более, чем всегда".

¹ Позже мы познакомимся со стандартной макрокомандой **invoke** для вызова процедур, в ней перенос на новую строку можно делать также и по символу запятой, например:

```
invoke MyProc, Param1,
      Param2, Param3
```

² Тем программистам, которым почему-то очень нужно начинать свои собственные имена с точки, могут сделать это, задав опцию компилятора **option DOTNAME**.

6.5. Пример полной программы на Ассемблере

Пишите программы в первую очередь для людей, и только потом для машин.

Стивен Макконнелл

Любой дурак сможет написать код, который поймет машина. Хорошие программисты пишут код, который сможет понять человек.

Мартин Фаулер

Прежде, чем написать первую полную программу на Ассемблере, необходимо научиться выполнять операции ввода/вывода, без которых, естественно, ни одна сколько-нибудь серьёзная программа обойтись не может. В самом языке машины, в отличие от, например, языка нашей учебной машины УМ-3, нет хороших команд ввода/вывода,¹ поэтому для того, чтобы, например, ввести целое число, необходимо выполнить некоторый достаточно сложный фрагмент программы на машинном языке.

Для организации ввода/вывода в наших примерах будет использоваться специально написанный набор макрокоманд. Вместо каждой макрокоманды Ассемблер будет при компиляции подставлять соответствующий этой макрокоманде набор предложений языка Ассемблер (так называемое *макро-расширение*). Таким образом, на первом этапе изучения языка Ассемблера, Вы избавляетесь от трудностей, связанных с организацией ввода/вывода, используя для этого заранее заготовленные фрагменты программ. Имена макрокоманд для наглядности будут выделяться **жирным** шрифтом, хотя это и не совсем правильно, так как они не являются служебными (зарезервированными) словами языка Ассемблер.

6.5.1. Макрокоманды консольного приложения

Если ничто другое не помогает, прочтите, наконец, инструкцию.

Прикладная Мерфология

При запуске программы в Windows ей для работы обычно предоставляется диалоговое окно, если это текстовое окно, то оно называется *консолью*. Ассемблерные программы, описанные в этой книге, будут запускаться в таком окне консоли, такие программы называются *консольными приложениями* Windows. В отличие от графических окон, программа в Windows может иметь только одну консоль.

Текст программы на Ассемблере как правило набирается в каком-нибудь текстовом редакторе (например, NotePad) в кодировке Windows (CP1251), а вот консольное окно по умолчанию "считает", что текст в него выводится в кодировке DOS (CP866). Латинские буквы в этих кодировках совпадают, а вот русские нет. Макрокоманды работы с консольным окном при необходимости сами в нужных местах автоматически перекодируют текст перед выводом. В том случае, если программа с русским текстом для вывода в консоль уже набрана в кодировке DOS (такие текстовые редакторы, конечно, есть), то нужно выключить автоматическую перекодировку, выполнив макрокоманду **ConsoleMode**. Повторное выполнение этой же макрокоманды снова включает режим перекодирования.

Это же относится и ко *вводу* русского текста, он поступает из консоли в кодировке DOS, перед выводом введённого русского текста на экран тоже следует отключить перекодировку (а потом не забыть снова включить её). Полностью перейти на набор ассемблерных программ в кодировке DOS тоже не панацея, так как необходимо, чтобы тексты для вывода в привычные пользователям окна сообщений (Message Box), наоборот, поступали в кодировке Windows 😊. Таким образом, к сожалению, здесь приходится "сидеть на двух стульях", с этим нужно смириться.

Консоль является окном Windows, следовательно, имеет свойства, которые можно получить, щёлкнув правой кнопкой мышки по заголовку окна и выбрав пункт меню "Свойства". Желательно установить там комфортный для работы (достаточно большой) размер шрифта.

¹ В машинном языке есть только команды для обмена одним байтом, словом или двойным словом между регистром процессора и заданным в команде особым периферийным устройством компьютера (портом ввода/вывода), с этими командами Вы познакомитесь позже.

Консоль полностью аналогична текстовому экрану программ на языке Free Pascal, поэтому при программировании на Ассемблере удобно пользоваться средствами, привычными программисту на этом языке высокого уровня (предполагается, что учащийся имеет навык программирования на Паскале). Такие средства предоставляет набор макрокоманд с mnemonic именами, похожими на имена из языка Free Pascal, они реализуют многие стандартные средства ввода/вывода и управление программой.

Имя макрокоманды не является служебным именем, поэтому большие и маленькие буквы в них различаются, однако для удобства программирования при описании соответствующих макроопределений иногда объявлены *синонимы* таких имён, например, у имени макрокоманды **gotoxy** есть синонимы **gotoXY** и **GotoXY**.

В программах на Ассемблере полезны следующие макрокоманды.

- **Макрокоманда очистки окна консоли**

ClrScr

Она эквивалентна вызову процедуры **ClrScr** языка Free Pascal.

- **Макрокоманда задания заголовка окна консоли**

ConsoleTitle Title

Эта макрокоманда меняет заголовок консольного окна. Текст заголовка используется диспетчером задач (процессов) Windows для идентификации задачи в списке выполняемых приложений. В качестве параметра может использоваться как непосредственный операнд-текст, так и адрес текста, например:

```
.data
T db "Заголовок 'окна' консоли",0
.code
ConsoleTitle 'Don't use " symbol !'
ConsoleTitle offset T
```

Обратите внимание, что текст может заключаться как в двойные, так и в одинарные кавычки (апострофы). Оператор **offset** <имя переменной> вычисляет адрес переменной в памяти. При задании *адреса* текста в виде **offset** T сам текст T в памяти должен заканчиваться нулевым байтом, как это принято в языке C. Как и в Паскале, повторение в строке апострофа дважды задаёт *один* символ апострофа, то же самое относится и к двойной кавычке.

- **Макрокоманда вывода строки текста**

outstr[ln] op1

Эта макрокоманда выводит на экран строку текста, определяемую своим параметром. В качестве параметра может использоваться как непосредственный операнд-текст, так и адрес текста, например:

```
.data
T db "Hello, ""World"" !",0
.code
outstrln "Привет, мир!"
outstr offset T
mov eax,offset T
outstr eax
outstrln 'Don't do that!'
```

Обратите внимание, что текст может заключаться как в двойные, так и в одинарные кавычки (апострофы). В строку выводимого текста можно вставлять через запятые числовые коды любых символов, например:

```
outstr "Первая строка",13,10,"Новая строка = ",3Eh
```

выведет текст

```
Первая строка
Новая строка = >
```

Так можно вставлять в выводимую строку управляющие символы и символы, не имеющие графического представления, но учтите, что числовые коды символов не могут стоять в *начале* строки. Как и в Паскале, добавление к имени макрокоманды окончания **ln** приводит к переходу после вывода на начало новой строки.

- **Макрокоманда ввода строки**

inputstr buf,Len[,text]

В качестве первого параметра задаётся *адрес* буфера для ввода текста, а в качестве второго параметра – максимально допустимая длина вводимого текста в формате i32, r32 или m32. В качестве необязательного третьего параметра можно задать строку-приглашение к вводу, формат этого последнего параметра такой же, как и в макрокоманде **outstr**.

В буфер помещается текст из стандартного потока `stdin` длиной не более `Len` или до конца введённой строки. Текст в буфере будет заканчиваться нулевым символом, поэтому длина буфера должна быть по крайней мере на единицу больше `Len`.¹ Реальное число введённых символов (без учёта нулевого символа) возвращается на регистре `EAX`, например:

```
.data
    Buf db 128 dup (?)
    T    db "Текст не длиннее 127 символов=",0
.code
    inputstr offset Buf,128,offset T
    outwordln eax,, "Длина введённого текста="
```

Стоит напомнить, что русские буквы вводятся в кодировке DOS и при необходимости последующего вывода их не следует перекодировать из Windows в DOS, например:

```
.data
    Buf db 128 dup (?)
.code
    inputstr offset Buf,127,"Введите русский текст : "
    outstrln "Введён текст : "
    ConsoleMode ;           Отключить перекодировку в DOS
    outstrln offset Buf
    ConsoleMode ;           Включить перекодировку в DOS
```

- **Макрокоманда ввода символа с клавиатуры**²

```
inchar op1[,text]
```

где операнд `op1` может иметь формат `r8` или `m8`. Код (номер в алфавите) введённого символа записывается в место памяти, определяемое операндом. Эта макрокоманда эквивалентна оператору Паскаля для ввода одного символа `Read(op1)`. В качестве необязательного второго параметра можно задать строку-приглашение к вводу, формат этого последнего параметра такой же, как и в макрокоманде **outstr**.

Забегая вперёд надо сказать, что, если по внешнему виду операнда нельзя установить его тип (и, следовательно, размер в байтах), то Ассемблер требует явно указать тип оператором **ptr**, например:

```
.data
    x db ?
.code
    inchar x,"Введите один символ = "
    mov ebx,offset x; ebx:=(адрес x)
    inchar byte ptr [ebx]; нельзя просто [ebx]
```

Оператор **offset** <имя переменной> вычисляет адрес переменной в памяти. Потом об операторах **ptr** и **offset** будет говориться более подробно. Следует ещё раз напомнить, что русские буквы вводятся в кодировке DOS, что может вызвать трудности при их обработке в программе, например, при сравнении с символами в кодировке Windows:

```
.data
    x db ?
.code
    inchar x,"Введите русскую букву = "
    cmp x,'Ж'; НЕПРАВИЛЬНО x и 'Ж' в разных кодировках
    cmp x,DOS('Ж'); ПРАВИЛЬНО 'Ж' в кодировке DOS
```

¹ Действие этой макрокоманды похоже на ввод в строку в языке Free Pascal `var S:string[buf-1];`.

² Из курсов по языкам высокого уровня Вам должно быть, известно, что на самом деле ввод производится из стандартного входного потока (для Паскаля он называется `input`), а вывод – в стандартный выходной поток (для Паскаля `output`). Обычно поток `input` подключён к клавиатуре, а `output` – к экрану терминала, хотя возможны и другие подключения. Эти же соглашения действуют и в данном Ассемблере, хотя за стандартными потоками здесь, следуя языку C, закреплены имена `stdin`, `stdout` и `stderr` (поток для вывода сообщений об ошибках, обычно он тоже подключён к экрану).

Здесь макрофункция **DOS** (*op1*), где *op1* задаёт символ в формате *i8*, *r8* или *m8*, возвращает в регистре AL символ, преобразованный из кодировки Windows в кодировку DOS. Макрофункция **DOS** подставляет регистр AL вместо текста своего вызова.

Наряду с *макропроцедурой inchar* реализована и *макрофункция @inchar* ([*text*]), которая подставляет введенный символ (записанный на регистре AL), на место своего вызова, например:

```
.code
    cmp @inchar("Введите цифру или точку = "), '.'
    je  Tochka
    mov bl,@inchar()
```

Обратите внимание, что при вызове макрофункции надо всегда задавать (даже пустые) круглые скобки.

- **Макрокоманда ввода символа без эха и контроля**

Readkey [*text*]

Код (номер в алфавите) введенного символа возвращается в регистре AL. Эта макрокоманда эквивалентна вызову функции **AL:=ReadKey** языка Free Pascal. В качестве необязательного параметра можно задать строку-приглашение к вводу, формат этого последнего параметра такой же, как и в макрокоманде **outstr**. Стоит напомнить, что, как и в языке Free Pascal, символы из дополнительного алфавита поступают в виде двух символов: символа с номером ноль и символа из дополнительного алфавита. Например, при нажатии клавиши **F1** сначала **ReadKey** считает символ #0, а затем символ #59 (';').

Наряду с *макропроцедурой Readkey* реализована и *макрофункция @Readkey* ([*text*]), которая подставляет введенный символ (записанный на регистре AL), на место своего вызова, например:

```
.code
    cmp @readkey(), '.'
    je  Tochka
```

Обратите внимание, что при вызове макрофункции надо всегда задавать (даже пустые) круглые скобки.

- **Макрокоманда вывода символа на экран**

outchar [*ln*] *op1* [, *text*]

где операнд *op1* может иметь формат *i8*, *r8* или *m8*. Значение операнда трактуется как беззнаковое число, являющееся кодом (номером) символа в алфавите, этот символ выводится в текущую позицию экрана. В качестве необязательного второго параметра можно задать текстовую строку, которая выводится перед символом, формат этого последнего параметра такой же, как и в макрокоманде **outstr**.

Для задания кода символа удобно использовать символьную константу языка Ассемблер, например, 'A', тогда можно не задаваться вопросом о соответствии самих символов и их номеров в используемом алфавите. Такая константа преобразуется компилятором Ассемблера именно в код этого символа, т.е. конструкция 'A' полностью эквивалентна записи **ord('A')** языка Паскаль. Например, макрокоманда **outchar '*'** выведет символ звёздочки на место курсора. Другими словами, макрокоманда **outchar** эквивалентна оператору Паскаля для вывода одного символа **Write(op1)**. Как и в Паскале, добавление к имени макрокоманды окончания **ln** приводит к переходу после вывода на начало новой строки.

Следует внимательно относиться в *выводе* русских букв, *введенных* по макрокомандам **inchar** и **inputstr**, т.к. они введены в кодировке DOS и их не надо перед выводом перекодировать в кодировку Windows, например:

```
.data
    X db ?
.code
    inchar X,"Введите русскую букву = "
    ostr "Введена буква = "
    ConsoleMode ;           Отключить перекодировку в DOS
    outchar X
    ConsoleMode ;           Включить перекодировку в DOS
```

- **Макрокоманда ввода целого числа**

```
inint[ln] op1[,text]
```

В качестве первого операнда `op1` можно использовать форматы `r8`, `r16`, `r32`, `m8`, `m16` или `m32`. В качестве необязательного второго операнда можно задать строку-приглашение к вводу, формат этого последнего параметра такой же, как и в макрокоманде **outstr**.

Макрокоманда производит ввод из стандартного потока на место первого операнда любого целого значения из диапазона $-2^{31} \dots +2^{32}-1$ (этот диапазон является объединением диапазонов знаковых и беззнаковых чисел формата двойного слова **dd**). При выходе вводимого значения за максимальный диапазон двойного слова макрокоманда выдаёт (предупредительную) диагностику об ошибке:

```
** inint: Number too big:=MaxLongint, CF:=1 **
```

при этом операнду присваивается значение `7FFFFFFFh` (`MaxLongint`). Флаг `CF:=0` при правильном вводе и `CF:=1` при неправильном. В качестве побочного эффекта для *правильного* числа устанавливается флаг `ZF:=1`, если лексема введённого числа начиналась со знака "-", иначе `ZF:=0`.

В отличие от Паскаля, где ограничителем лексемы целого значения во входном потоке является символ пробела, табуляции или конец строки, макрокоманда **inint** дополнительно считает концом целой лексемы любой символ, не являющийся цифрой. В качестве побочного эффекта для такого "плохого" конца лексемы целого числа устанавливается флаг `SF:=1`, иначе `SF:=0`. Например, из строки `"-123ABCD"` введётся число `-123`, в во входном потоке останется строка `"BCD"`, при этом будет `SF:=1`.

В случае, если введённое число превышает размер операнда `op1`, то производится *усечение* этого числа без выдачи диагностики об ошибке.¹ Добавление к имени макрокоманды окончания **ln** приводит к очистке буфера ввода и эквивалентна вызову стандартной процедуры `Readln` языка Паскаль, например:

```
.data
  X dd ?
.code
  inint X, "Введите целое число X="
  jc   BigNum; Число вне диапазона  $-2^{31} \dots +2^{32}-1$ 
  jz   Minus; Введено отрицательное число
  js   BadEnd; Плохое окончание числа
```

Наряду с *макропроцедурой* **inint** реализована и *макрофункция* **@inint**([text]), которая подставляет введённое число (записанное на регистре `EAX`), на место своего вызова, например:

```
.code
  cmp @inint(), 0
  je   ZeroInput
  outintln @inint("Введите число="), , \
          "Введено число="
```

Обратите внимание, что при вызове макрофункции надо всегда задавать (даже пустые) круглые скобки.

Ограничения. Выводится фатальная диагностика при попытке ввода в регистры-указатели вершины стека **inint** (`e`)`sp`; макрокоманда работает неправильно при использовании регистра `ESP` как базового, например, **inint** `X[esp]`.

- **Макрокоманды вывода целого значения**

```
outint[ln] op1[, [op2][,text]]
outword[ln] op1[, [op2][,text]]
```

Здесь, как всегда при описании синтаксиса, квадратные скобки говорят о том, какие части макрокоманды можно опустить. В качестве первого операнда `op1` можно использовать форматы `r8`, `r16`, `r32`, `m8`, `m16`, `m32`, `m64` или `i32`, а в качестве необязательного второго – форматы `i8`, `r8` или `m8`. В качестве необязательного третьего параметра можно задать текстовую строку, которая выводится перед целым числом, формат этого последнего параметра такой же, как и в макрокоманде **outstr**.

Действие макрокоманды **outint** `op1[, [op2][,text]]` эквивалентно выполнению процедуры вывода одного знакового целого значения языка Паскаль `write([text,] op1[:op2])`, где па-

¹ Это эквивалентно работе программы на языке Free Pascal в режиме `{ $R- }`, т.е. без контроля выхода величины за допустимый диапазон. Обычно Free Pascal по умолчанию работает именно в этом режиме. При необходимости в программах на Ассемблере такой контроль надо проводить самому программисту, вводя число в переменную формата **dd** и затем проверяя нужный диапазон.

параметр `op2` задаёт ширину поля вывода. Макрокоманда **outint** имеет короткий синоним **OutI**. Действие же макрокоманды с именем **outword** отличается только тем, что первый операнд при выводе *трактуются* как беззнаковое (неотрицательное) целое число. Эта макрокоманда имеет синоним **OutUnsigned** или более коротко **OutU**. Как и в Паскале, добавление к имени макрокоманды окончания **ln** (или **Ln**) приводит к переходу после вывода на начало новой строки, например:

```
.data
    x    dd    ?
.code
    inint x
    outintln x,, "Введено x="
    OutU -1,, "MaxLongWord="
```

- **Макрокоманда перехода на новую строку**

newline [n]

предназначена для перевода курсора к началу следующей строки экрана и эквивалентна вызову n раз стандартной процедуры без параметров `Writeln` языка Паскаль. По умолчанию n=1.

- **Макрокоманда без параметров**

flush

предназначена для очистки буфера ввода и эквивалентна вызову стандартной процедуры без параметров `Readln` языка Паскаль.

Ограничение. К сожалению, пока макрокоманда работает неправильно, если буфер ввода уже совсем пуст (не содержит конечного служебного символа LF, уже введённого, например, по **inchar**).

- **Макрокоманда перемещения курсора**

GotoXY x, y

Параметры x и y могут быть форматов r8, r16, m8, m16 или i8. Действие этой макрокоманды эквивалентно выполнению оператора процедуры `GotoXY(x, y)` языка Free Pascal. При выходе x или y за границы окна макрокоманда игнорируется.

- **Макрокоманда X-позиции курсора**

WhereX

Номер столбца позиции курсора (1..80) в окне консоли возвращается в регистре AL. Эта макрокоманда эквивалентна вызову функции `AL:=WhereX` языка Free Pascal.

- **Макрокоманда Y-позиции курсора**

WhereY

Номер строки позиции курсора в окне консоли возвращается в регистре AL. Эта макрокоманда эквивалентна вызову функции `AL:=WhereY` языка Free Pascal. Обычно окно консоли создаётся длиной 50 строк (с возможностью прокрутки текста). Для уменьшения размеров окна консоли можно использовать макрокоманду **SetScreenSize**, например, задание окна стандартного размера:

SetScreenSize 80,25

- **Макрокоманда смены текущих цветов фона и символов**

SetTextAttr [colors]

Цвета в палитре из 16 цветов, как и в языке Free Pascal, задаются в параметре формата i8 в виде 16*<цвет фона>+<цвет букв>

Определены имена следующих констант цветов:

Black	= 0h
Blue	= 1h
Green	= 2h
Red	= 4h
Bright	= 8h
DarkGray	= Bright
Cyan	= Blue+Green
Brown	= Green+Red
Magenta	= Blue+Red
LightMagenta	= Bright+Blue+Red
LightGray	= Blue+Green+Red
LightBlue	= Bright+Blue
LightGreen	= Bright+Green

```

LightRed      = Bright+Red
Yellow        = Bright+Green+Red
White         = Bright+Blue+Green+Red

```

При отсутствии параметра подразумевается цвет `LightGray`. Действие макрокоманды эквивалентно (за исключением случая пустого параметра) оператору присваивания языка Free Pascal

```
TextAttr := colors
```

Примеры:

```

SetTextAttr 16*Blue+Yellow
OutStrLn "Цвет 16*Blue+Yellow"
SetTextAttr Blue
OutStrLn 'Цвет 16*Black+Blue'
SetTextAttr
OutStrLn 'Цвет 16*Black+LightGray'

```

- **Макрокоманда вывода окна сообщения**

```
MsgBox Caption, Message, Style
```

Эта макрокоманда выводит на экран (поверх остальных окон) небольшое окно сообщения с заголовком `Caption`, текстом сообщения `Message` и кнопками ответов. Пиктограмма окна сообщений, количество и вид кнопок определяется стилем окна `Style`. Стилль окна определяется набором (суммой) битовых признаков, которым присвоены следующие мнемонические имена:





```

MB_OK (кнопка "ОК"),
MB_OKCANCEL (кнопки "ОК" и "Отмена"),
MB_ABORTRETRYIGNORE (кнопки "Прервать", "Повтор" и "Пропустить"),
MB_YESNOCANCEL (кнопки "Да", "Нет" и "Отмена"),
MB_YESNO (кнопки "Да" и "Нет"),
MB_RETRYCANCEL (кнопки "Повтор" и "Отмена"),
MB_CANCELTRYCONTINUE (кнопки "Отмена", "Повторить" и "Продолжить").

```

Когда стиль опущен, предполагается `MB_OK`. Пиктограмма окна задаётся битовым признаком:

```

MB_ICONSTOP ( , MB_ICONQUESTION ( ,
MB_ICONEXCLAMATION ( , обычно здесь выдаётся ещё и звуковой сигнал),
MB_ICONASTERISK ( , MB_USERICON (без пиктограммы).1

```

При нажатии на кнопку в регистре `EAX` возвращается код этой кнопки, мнемонические имена этих кодов:

```

IDOK, IDCANCEL (также, если нажать клавишу ESC или крестик закрытия окна),
IDABORT, IDRETRY, IDIGNORE, IDYES, IDNO, IDTRYAGAIN, IDCONTINUE


```

При выводе окна по умолчанию выбирается (если просто нажать `ENTER`) первая по порядку из выведенных кнопок, чтобы выбрать по умолчанию другую кнопку, надо в параметре `Style` задать битовый признак с номером кнопки в окне

```
MB_DEFBUTTON{1,2,3}
```

В качестве параметров `Caption` и `Message` можно использовать как непосредственный операнд-текст, так и адрес текста, например:

```

.data
T1 db 'Текст заголовка окна', 0
T2 db "Было хорошо ?", 0
.code
MsgBox offset T1, "Привет, мир!", MB_OK+MB_ICONEXCLAMATION
MsgBox 'Заголовок окна', offset T2, \
      MB_YESNO+MB_ICONQUESTION+MB_DEFBUTTON2; Пессимист 
cmp eax, IDNO
je Bad_Chance

```

- **Макрокоманда задания паузы**

```
pause [text]
```

В качестве необязательного параметра можно использовать как непосредственный операнд-текст, так и адрес текста, например:

¹ В этом случае можно задать свою пиктограмму в так называемом файле ресурсов, первая пиктограмма в этом файле (он имеет расширение `.ico`) автоматически будет также пиктограммой заголовка консоли и выполняемого файла `.exe` в проводнике Windows.

pause "Для продолжения нажмите любую клавишу..."

- **Макрокоманда порождения динамической переменной**

new size; size≥0

Макрокоманда порождает динамическую переменную размером size (формата i32) и возвращает адрес этой переменной в регистре EAX. При исчерпании кучи возвращается значение

nil equ 0

Например:

```
Node struct
    Left dd ?
    Right dd ?
    Key dd ?
Node ends
.code
new sizeof node
mov [eax].Node.Left, nil
```

Размер выдаваемой макрокомандой **new** динамической памяти округляется вверх до величины, кратной 16 байт (для **new 0** будет **new 16**) и выравнивается на границу 16 байт. Макрокоманда

HeapBlockSize address

возвращает на регистре EAX размер блока памяти с адресом address, выделенной макрокомандой **new**.

- **Макрокоманда уничтожения динамической переменной**

dispose address

Макрокоманда уничтожает динамическую переменную, адрес которой имеет формат r32 или m32. Как и в Паскале, значение переменной address при этом не меняется, например:

```
new sizeof node
dispose eax
```

Повторное уничтожение динамической переменной не рассматривается как ошибка.

- **Макрокоманда без параметров**

TotalHeapAllocated

Эта макрокоманда возвращает на регистре EAX общее количество динамической памяти, выделенной программе к этому моменту по макрокомандам **new**, но ещё не уничтоженной по макрокомандам **dispose**, это может использоваться для контроля утечки памяти. Данный механизм контроля утечки памяти работает правильно, только если все динамические переменные в программе одного размера.

- **Макрокоманда вывода даты**

OutDate[ln] [text]

Эта макрокоманда выводит текущую дату в формате dd.mm.yyyy, если задан параметр, то он выводится перед датой, например:

```
.data
T db "Сегодня ", 0
.code
OutDateIn "Текущая дата = "
OutDate offser T
```

- **Макрокоманда вывода текущего времени**

OutTime[ln] [text]

Эта макрокоманда выводит текущее время в формате hh:mm:ss, если задан параметр, то он выводится перед временем, например:

```
.data
T db "Сейчас ", 0
.code
SetTextAttr Yellow
OutTimeIn "Текущее время = "
SetTextAttr
OutTime offser T
```

- **Макрокоманда перехода на новую программу**

RunExe FileName

Запуск в этом же консольном окне новой программы Windows с именем файла FileName. В качестве параметра может использоваться как непосредственный операнд-текст, так и адрес текста, например:

```
.data
    ProgName db "MyProg.exe Param1 Param2",0
.code
    RunExe "MyGoodProgram.exe"
    RunExe offset ProgName
```

После завершения вызванной программы производится возврат в старую программу, которая получает назад своё консольное окно со всеми изменениями (заголовок, цвета, позиция курсора и т.д.), возможно сделанными вызванной программой. Вызванная консольная программа, однако, может открыть и своё собственное консольное окно, вызвав макрокоманду **NewConsole**, в этом случае после возврата старая программа продолжит выполнение в своём неизменённом окне. Кроме того, вызванная программа может быть и графическим приложением Windows, например:

```
RunExe "C:\ Windows\System32\calc.exe"
```

- **Макрокоманда выхода из программы**

exit

Действие этой макрокоманды эквивалентно выходу на завершающей **end**. в программе на Паскале. Макрокоманда **exit** немедленно закрывает консольное окно, поэтому при запуске программы на счёт из загрузочного файла .exe перед этой макрокомандой необходимо поставить либо задержку окончания работы **pause**, либо выдавать окно сообщений **MsgBox**, иначе результаты работы программы сразу перестают быть видимы. При запуске на счёт из пакетного файла `makeit.bat` этого можно не делать, так как команда **pause** стоит в самом этом пакетном файле.

6.5.2. Структура программы на Ассемблере

Большинство программ на сегодняшний день подобны египетским пирамидам из миллиона кирпичиков друг на друге и без конструктивной целостности – они просто построены грубой силой и тысячами рабов.

Алан Кёртис Кэй

Итак, программа является набором предложений Ассемблера, являющихся командами, макрокомандами, директивами и предложениями резервирования памяти. По принципу Фон Неймана в любом месте памяти может находиться как команда, так и переменная, однако для удобства программирования лучше записывать команды и различные по смыслу данные в разных *секциях* программы. Так, в секции с заголовком **.const** лучше хранить числовые и текстовые *константные* переменные программы, по умолчанию эта секция при выполнении программы закрывается на запись, что позволяет предотвратить случайное изменение таких констант, например:

```
.const
    T1 db "Ответ X= "
    B dw -8; это параметр, заданный программистом
```

В секции с заголовком **.data** лучше хранить переменные с начальными значениями, которые могут далее меняться при счёте программы, например:

```
.data
    x dw -1; переменная x может меняться в программе
    db 1; безымянная переменная с начальным значением
    y dd 100 dup (0); массив из 100 изначально нулевых элементов
```

Наоборот, в секции с заголовком **.data?** нужно описывать так называемые *неинициализированные* переменные, не имеющие начальных значений (в соответствии со стандартом Паскаля). Начальные значения у переменных в этой секции при трансляции игнорируются и вызывают предупредительную диагностику, например:

```
.data?
    a dd ?; переменная с неопределённым начальным значением
    db ?; безымянная переменная без начального значения
```

```

z    dw 200000 dup (?); массив из 200000 элементов
Bad dw 1; ПРЕДУПРЕЖДЕНИЕ (warning), в секции будет Bad dw ?

```

Выделение для неинициализированных переменных отдельной секции позволяет не хранить эту "пустую" секцию в программе, что позволяет уменьшить его размер в файловой системе ("на диске"). Вообще говоря, этим можно пренебречь и хранить неинициализированные переменные вместе с инициализированными в секции **.data**. В секции **.data** можно задавать и предложения-команды, в этом случае они тоже трактуются как (безымянные) области памяти с начальными значениями, например:

```

.data
c    dw -1
      mov eax, 1; безымянная область памяти со значением команды 1
q    dd 10 dup (-1)

```

Обычно задание команд в секции данных не имеет большого смысла и используется крайне редко. Отметим, что по умолчанию секции констант и данных закрыты на выполнение команд, поэтому передать в них управление и использовать константы и переменные в качестве команд просто так нельзя. И, наконец, в секции с заголовком **.code** записываются команды (и макрокоманды) программы, по умолчанию эта секция при счёте закрывается на запись, что позволяет предотвратить случайное изменение команд,² например:

```

.code
outstr "Начало программы"
mov    eax, 1
outintln eax

```

В секции кода не запрещается указывать и переменные (как с начальным значением, так и неинициализированные). Обычно это делается, когда в языке Ассемблера невозможно явно задать какую-нибудь команду машины. Например, так можно использовать команду-префикс смены длины регистра с кодом 66h (вообще-то эту команду-префикс ставит сам Ассемблер):

```

mov    eax, 1; выполняется как mov    eax, 1
db     66h;   префикс смена длины регистра с 32-х на 16 бит
mov    eax, 1; выполняется как mov    ax, 1
mov    ax, 1; здесь Ассемблер сам вставляет команду-префикс 66h

```

Начало описания новой секции отмечает конец предыдущей, последняя секция закрывается директивой конца модуля **end**. Допускается задавать в модуле на Ассемблере несколько одноимённых секций, при этом каждая следующая секция Ассемблером объединяется (добавляется в конец) предыдущей такой же секции.

Вообще говоря, у каждой секции есть ещё и *имя*, например, секция **.code** по умолчанию имеет имя **_TEXT**, а секция **.data** имя **_DATA**. Под такими именами секции фигурируют в листинге программы и в паспортах объектных модулей. Считается, что каждая секция является сокращённым описанием *сегмента*. Например, секция **.data** "разворачивается" компилятором Ассемблера в такое описание сегмента:

```

_DATA segment para public 'DATA'
. . .
_DATA ends

```

Это называется *стандартными* директивами сегментации, а заголовок сегмента **.data** соответственно *упрощённой* формой сегментации. Значение параметров в директиве **segment** объясняются в главе, посвященной описанию работы редактора внешних связей.

Рассмотрим теперь пример простой *полной* программы на Ассемблере. Эта программа должна вводить значение целой переменной **A** и реализовывать оператор присваивания (в смысле языка Паскаль) и вывести значение переменной **X**.

```

X := (2*A - (A+B)2) mod 7

```

¹ Исключение составляют команды перехода, их в секциях данных указывать нельзя.

² Секцию команд можно закрыть и на чтение (команд как чисел на регистры арифметико-логического устройства), однако так обычно не делается, так как в этой секции часто хранят и различные *константы*. Заметим, что у программиста на Ассемблере, в соответствии с принципами Фон Неймана, имеется возможность открыть секции команд и констант на запись, а секции данных на выполнение. Как это сделать описано в главе, посвященной модульному программированию.

Пусть A, B и X – *знаковые* целые переменные, описанные таким образом:

```
A dd ?  
B dw -8; это параметр, заданный программистом  
X db ?
```

Параметр, в отличие от "настоящей" константы, иногда, хотя и редко, может меняться, например, перед запуском задачи на счёт с другими входными данными. На языке Free Pascal нужная программа выглядит, например, таким образом:

```
program P(input,output);  
  var A: Longint; B: Integer=-8; X: Shortint;  
begin  
  Write('Введите целое A='); Read(A);  
  X := (2*A - sqr(A+B)) mod 7;  
  Writeln('Ответ X=',X)  
end.
```

В переменной X будет получаться результат работы. Это *короткое* целое число (остаток от деления на 7, который помещается в один байт).

В начале программы на Ассемблере задаются предложения-директивы, устанавливающие режимы работы. Директива

```
include console.inc
```

вставляет в начало программы текстовый файл с именем console.inc, содержащий все директивы и макроопределения, необходимые для правильной работы. Далее следуют необходимые для работы секции программы (подробные пояснения будут даны сразу же вслед за текстом этого примера):

```
include console.inc  
comment *  
  директива include вставляет в начало программы  
  текст из файла с директивами и макроопределениями  
*  
.const  
  B dw -8  
.data?  
  A dd ?  
  X db ?  
.code  
; X := (2*A - (A+B)2) mod 7  
start:  
  inint A, 'Введите целое A='  
  mov ebx,A; ebx := A  
  mov ax,B; ax := B  
  cwd; eax := Longint(B)  
  add eax,ebx; eax := B+A=A+B  
  imul eax; <edx,eax> := (A+B)2  
; Пусть (A+B)2 помещается в eax !  
  add ebx,ebx; ebx := 2*A  
; Пусть 2*A помещается в ebx !  
  sub ebx,eax; ebx := 2*A - (A+B)2  
; Пусть 2*A - (A+B)2 помещается в ebx !  
  mov eax,ebx; делимое  
  cdq; <edx,eax> := 64-битное делимое  
  mov ebx,7; 32-битный делитель  
  idiv ebx; edx := (2*A - (A+B)2) mod 7  
  mov X,dl; ответ длиной в байт  
  outintln X, "Ответ X="  
  exit; выход из программы  
end start
```

Подробно прокомментируем текст этой программы. В начале секции кода указана метка start, она определяет *первую* выполняемую команду программы. Следующие два предложения выводят приглашение для ввода переменной A и вводят значение этой переменной.

Далее идет непосредственное вычисление правой части оператора присваивания. Задача усложняется тем, что величины А и В имеют разную длину (4 и 2 байта соответственно) и непосредственно складывать их нельзя. Приходится командами

```
mov ax,B; ax := B
cld;      eax := Longint(B)
```

преобразовать короткое *знаковое* целое число В, которое считалось в регистр АХ, в сверхдлинное целое на регистре ЕАХ (можно обойтись и одной командой `movsx eax,B`). Далее вычисляется значение выражения $(A+B)^2$. Произведение занимает два регистра (регистровую пару) <ЕДХ, ЕАХ>. Пока для простоты предполагается, что в старшей части произведения (регистре ЕДХ) находится *незначащая* часть произведения и в качестве результата можно взять только значение регистра ЕАХ. Далее вычисляется делимое $2*A - (A+B)^2$, оно пересылается на регистр ЕАХ, и можно приступить к вычислению остатка от деления на 7.

Так как делимое является 32-битным числом, то на первый взгляд можно было бы использовать команду целочисленного деления 32-битного числа на 16-битное, например, так

```
mov bx,7; 16-битный делитель
idiv bx;   ax := <dx,ax> div 7; dx := <dx,ax> mod 7
```

Здесь, однако, очень часто возникает серьёзная ошибка. Остаток от деления <ДХ, АХ> `mod 7` *всегда* поместиться в отведённое ему место (регистр ДХ), так как он по абсолютной величине меньше 7. В то же время частное <ДХ, АХ> `div 7` может не поместиться в регистр АХ, что приведёт к аварийному завершению программы. Исходя из этого следует обязательно использовать команду деления 64-битного делимого на 32-битный делитель

```
cdq;      <edx,eax> := 64-битное делимое
mov ebx,7; 32-битный делитель
idiv ebx;   edx := (2*A - (A+B)2) mod 7
```

Вот теперь ответ из регистра ЕДХ можно присвоить переменной Х, а так как Х имеет длину всего один байт, то берётся только значение регистра dl – самой младшей части регистра ЕДХ.

Последним предложением в секции кода является макрокоманда

```
exit
```

Эта макрокоманда заканчивает выполнение нашей программы, она эквивалентна выходу программы на Паскале на конечный **end**.

Затем следует очень важная директива конца программного модуля на Ассемблере

```
end start
```

Обратите внимание на параметр этой директивы – метку *start*. Она указывает *входную точку* программы, т.е. её первую выполняемую команду. Явное задание входной точки позволяет начать выполнения секции команд с любого места, а не обязательно с её первой команды. Такая входная точка может указываться только в головном модуле программы, где находится первая выполняемая команда. Остальные модули на Ассемблере (если они есть) должны оканчиваться директивой **end** без параметра.

Сделаем теперь важные замечания к этой программе. Во-первых, не проверялось, что команды сложения и вычитания дают правильный результат (для этого, как Вы знаете, после выполнения этих команд было бы необходимо проверить флаг переполнения OF, т.к. наши целые числа *знаковые*). Во-вторых, команда умножения располагает свой результат в двух регистрах <ЕДХ, ЕАХ>, а в написанной программе результат произведения брался из регистра ЕАХ, т.е. предполагалось, что на регистре ЕДХ находятся только *незначащие* биты произведения. По-хорошему надо было бы после команды умножения проверить условие, что флаги OF=CF=0, это гарантирует, что в ЕДХ содержатся только нулевые биты, если $AX \geq 0$, и только двоичные '1', если $AX < 0$. Другими словами, все биты в регистре ЕДХ должны совпадать со старшим битом в регистре ЕАХ, для *знаковых* чисел это и есть признак того, что в регистре ЕДХ содержится *незначащая* часть произведения. В *учебных* программах такие проверки делаются не всегда, но в "настоящих" программах, которые Вы будете создавать на компьютерах, эти проверки являются обязательными. Далее будет показано, как это делать, создавая *надёжные* программы.

Теперь, после изучения арифметических операций, перейдём к рассмотрению команд *переходов*, которые понадобятся нам для программирования условных операторов и циклов. Напомним, что по-

сле освоения материала этой книги Вы должны уметь отображать на Ассемблер большинство конструкций языка Паскаль.

6.6. Переходы

Выходов нет, есть только переходы.

Перед переходом сначала посмотрите налево, потом направо.

В большинстве компьютеров, в том числе и в изучаемом сейчас, по принципу Фон Неймана реализовано *последовательное выполнение команд*. В соответствии с этим принципом, перед выполнением текущей команды счётчик адреса устанавливается на следующую (ближайшую с большим адресом) команду в оперативной памяти.¹ Таким образом, команды программы, расположенные последовательно в оперативной памяти, выполняются друг за другом. Однако, как и в учебной машине УМ-3, такой порядок выполнения программы может изменяться с помощью команд *переходов*, при этом следующая команда может быть расположена в другом месте оперативной памяти. Ясно, что без переходов компьютеры функционировать не могут: скорость процессора так велика, что он может быстро "по одному разу" выполнить все команды в оперативной памяти. Кроме того, большинство алгоритмов содержат внутри себя циклы и условные операторы, которые также реализуются с помощью команд переходов.

Понимание механизма выполнения переходов очень важно при изучении архитектуры ЭВМ, это позволяет уяснить логику работы процессора и лучше понять архитектуру нашего компьютера. Все переходы можно разделить на два вида.

- Переходы, вызванные выполнением процессором специальных *команд переходов*.
- Переходы, которые *автоматически* выполняет процессор при наступлении определённых событий в центральной части компьютера или в его периферии (устройствах ввода/вывода).

Начнём рассмотрение переходов для компьютеров изучаемой нами архитектуры. Напомним, что в плоской модели памяти логический адрес начала следующей выполняемой команды хранится в регистре-счётчике адреса EIP.

Следовательно, для осуществления любого перехода необходимо в этот регистр EIP занести новое значение, соответствующее месторасположению следующей выполняемой команды. По способу изменения значения регистра EIP различают *относительные* и *абсолютные* переходы. При относительном переходе происходит *знаковое* сложение содержимого регистра EIP с некоторой величиной, например,

$$EIP := (EIP \pm Value) \bmod 2^{32}$$

При абсолютном переходе происходит просто присваивание регистру EIP нового значения:

$$EIP := Value$$

Далее, относительные переходы будут классифицироваться по величине той константы Value, которая прибавляется к значению счётчика адреса EIP. При *коротком* переходе величина этой *знаковой* константы не превышает по размеру одного байта (напомним, что она обозначается i8, т.е. лежит в диапазоне от -128 до +127):

$$EIP := (EIP + i8) \bmod 2^{32},$$

а при *длинном* переходе эта *знаковая* константа имеет размер i32 (двойное слово):

$$EIP := (EIP + i32) \bmod 2^{32}$$

Легко понять, что абсолютные переходы делать короткими бессмысленно, так как они могут передавать управление только в самое начало (при $i8 \geq 0$) или в самый конец (при $i8 < 0$) оперативной памяти, эти участки, как уже упоминалось, в плоской модели недоступны программе пользователя.

Следующей основой для классификации переходов будет месторасположение величины, используемой при *абсолютном* переходе для задания нового значения регистра EIP. При *прямом* переходе эта величина является просто числом-адресом команды (здесь это *непосредственный* адрес в са-

¹ Однако, как уже говорилось, при достижении в программе конца оперативной памяти следующей будет выполняться команда, расположенная в начале памяти, т.е. память как бы замкнута в кольцо. Заметим, что в плоской модели памяти, первые и последние 64 Кб оперативной памяти "изъяты из обращения" и программам пользователя они никогда не выделяются, так что такой "переход по кольцу" невозможен.

мой команде `i8` или `i32`). При *косвенном* переходе нужная величина располагается в памяти или на регистре, а в команде перехода задаётся *адрес* той области памяти (или *номер* этого регистра), откуда и будет извлекаться необходимое число, например:

```
EIP := [m32]      или      EIP := r32
```

Здесь на регистр `EIP` будет заноситься число, содержащееся в четырех байтах памяти по адресу `m32`, или из регистра `r32`, т.е. в приведенной классификации это *длинный абсолютный косвенный* переход.¹

Таким образом, каждый переход можно классифицировать по его свойствам: относительный – абсолютный, короткий – длинный, прямой – косвенный. Разумеется, не все из этих переходов реализуются в архитектуре компьютера, так, уже говорилось, что короткими бывают только относительные переходы, кроме того, относительные переходы бывают только прямыми.

6.7. Команды переходов

Не уверен – не *goto* !

Здесь будут изучены *команды* переходов. Заметим, что эти команды предназначены *только* для передачи управления в другое место программы, они *не меняют* никаких флагов.

6.7.1. Команды безусловного перехода

... *goto* должен быть изгнан отовсюду,
кроме – быть может – простого языка
машины.

Э. Дейкстра

Рассмотрим сначала команды безусловного перехода, которые всегда передают управление в указанную в них точку программы. На языке Ассемблера все эти команды записываются в виде

```
jmp op1
```

Здесь операнд `op1` может иметь форматы, показанные в таблице 6.1.¹

Таблица 6.1. Форматы операнда команды безусловного перехода.

op1	Способ выполнения	Вид перехода
i8	EIP := (EIP + Longint (i8)) mod 2 ³²	относительный короткий прямой
i32	EIP := (EIP + i32) mod 2 ³²	относительный длинный прямой
r16	EIP := Longint ([r16])	абсолютный длинный косвенный
r32	EIP := [r32]	абсолютный длинный косвенный
m16	EIP := Longint ([m16])	абсолютный длинный косвенный
m32	EIP := [m32]	абсолютный длинный косвенный

Рассмотрим теперь, как на языке Ассемблера задаётся код операции и операнд команд безусловного перехода. Для указания близкого относительного перехода в команде обычно записывается метка (т.е. имя) команды, на которую необходимо выполнить переход, например:

```
jmp L; Перейти на команду, помеченную меткой L
```

Напомним, что вслед за меткой команды, в отличие от метки области памяти, ставится двоеточие. Так как значением метки является её смещение в оперативной памяти, то компилятору Ассемблера приходится самому вычислять это смещение `i8` или `i32`, которое необходимо записать на место операнда в команде на машинном языке, например:

```
L: add ebx, ebx;
    . . .
    . . .
    . . .
    jmp L; L=i8 или i32
```

← i8 или i32 (со знаком!)

Необходимо также учитывать, что в момент выполнения команды перехода счётчик адреса `EIP` уже указывает на *следующую* команду, что, конечно, существенно при вычислении величины сме-

¹ Значение косвенного перехода может быть и 16-тиразрядным, т.е. иметь форматы `m16` или `r16`. В этом случае производится знаковое расширение величины перехода до 32-х бит. Как уже говорилось, в плоской модели памяти это не имеет смысла, так как адрес перехода почти всегда попадает в запрещённые области первых и последних 64 Kb оперативной памяти.

щения в команде относительного перехода. Однако, так как эту работу выполняет компилятор Ассемблера при трансляции программы, на такую особенность не будем пока обращать внимания, она будет существенной далее, при изучении команд вызова процедуры и возврата из процедуры. Заметим, что программист и сам может вычислить нужное смещение (i8 или i32) и явно задать его в команде перехода в виде:

```
jmp $+D; Здесь D – знаковая величина формата i8 или i32, например  
jmp $-8; EIP := EIP-8-⟨длина команды jmp⟩
```

Здесь символом \$ обозначено текущее значение так называемого счётчика размещения (location counter), это адрес первого байта текущей команды в секции кода или адрес первого байта текущей переменной в секциях данных. Таким образом, для секции кода во время счёта $\$ = \text{EIP} - \langle \text{длина текущей команды} \rangle$, поэтому выполнение команды `jmp $` есть лучший способ заикнуться 😊.

При близком переходе формат для операнда L (i8 или i32) выбирается компилятором Ассемблера автоматически, в зависимости от расстояния в байтах между командой перехода и командой с указанной меткой.

Заметьте, что, если в команде перехода задаётся метка, то при прямом переходе она описана в программе с двоеточием (это метка команды), а при косвенном – без двоеточия (это метка области памяти). Отметим здесь также одно важное преимущество относительных переходов перед абсолютными переходами. Значение i8 или i32 в команде относительного перехода зависит только от расстояния в байтах между командой перехода и точкой, в которую производится переход. При любом изменении в секции кода *вне* этого диапазона команд значения i8 или i32 не меняются. Это полезное свойство относительных переходов позволяет, например, при необходимости достаточно легко *склеивать* две секции кода из разных программных модулей, что используется в системной программе редактора внешних связей (эта программа будет изучаться позже). Как видно, архитектура изучаемого компьютера обеспечивает большой спектр команд безусловного перехода, вспомним, что в учебной машине УМ-3 была только одна такая команда. На этом закончим краткое рассмотрение команд безусловного перехода.

6.7.2. Команды условного перехода

Хороший язык программирования помогает программистам писать хорошие программы. Ни один из языков программирования не может запретить своим пользователям писать плохие программы.

Киз Костер

Все команды условного перехода в изучаемой архитектуре выполняются по схеме, которую на Паскале можно записать как

```
if <условие перехода> then goto L
```

и производят *относительный прямой* переход, если выполнено некоторое условие перехода, в противном случае продолжается последовательное выполнение команд программы. На Паскале такой переход чаще всего задают в виде условного оператора:

```
if op1 <отношение> op2 then goto L
```

где <отношение> – один из знаков операций отношения = (равно), <> (не равно), > (больше), < (меньше), <= (меньше или равно), >= (больше или равно). Если обозначить $\text{rez} = \text{op1} - \text{op2}$, то этот оператор условного перехода Паскаля можно записать в эквивалентном виде сравнения с нулём:

```
if rez <отношение> 0 then goto L
```

В изучаемой архитектуре все машинные команды условного перехода, кроме одной, вычисляют условие перехода, анализируя один, два или три флага из регистра флагов, и лишь одна команда условного перехода вычисляет условие перехода, анализируя значение регистра `ecx`.¹

Команда условного перехода в языке Ассемблера имеет такой вид:

```
j<мнемоника перехода> op1
```

Здесь op1 может иметь форматы i8 или i32¹, и команда реализует *относительный прямой* переход и выполняется по схеме

¹ Есть и команда условного перехода (с префиксом `b6h`), по регистру `CX`, но она редко используется.

$EIP := (EIP + Longint(op1)) \bmod 2^{32}$

Мнемоника перехода (это от одной до трёх букв) связана со значением анализируемых флагов (или регистра `ecx`), либо со *способом формирования* этих флагов. Чаще всего программисты формируют нужные флаги, проверяя отношение между двумя операндами `op1 <отношение> op2`, для чего выполняется команда *вычитания* или команда *сравнения*. Команда сравнения имеет мнемонический код операции **cmp** и такие же допустимые форматы операндов, как и команда вычитания:

cmp `op1, op2`

Она и выполняется точно так же, как команда вычитания за исключением того, что разность *не записывается* на место первого операнда. Таким образом, единственным результатом команды сравнения является формирование флагов, которые устанавливаются так же, как и при выполнении команды вычитания. Вспомните, что программист может *трактовать* результат вычитания (сравнения) как производимый над *знаковыми* или же *беззнаковыми* числами. Как уже известно, от этой трактовки зависит и то, будет ли один операнд больше другого или же нет. Так, например, рассмотрим два коротких целых числа `0FFh` и `01h`. Как *знаковые* числа `0FFh` = -1 меньше `01h` = 1, а как *беззнаковые* числа `0FFh` = 255 больше `01h` = 1.

Исходя из этого, принята следующая терминология: при сравнении *знаковых* целых чисел первый операнд может быть *больше* (greater) или *меньше* (less) второго операнда. При сравнении же *беззнаковых* чисел будем говорить, что первый операнд *выше* (above) или *ниже* (below) второго операнда. Ясно, что действию "выполнить переход, если первый операнд больше второго" будут соответствовать разные машинные команды, если трактовать операнды как знаковые или же беззнаковые. Можно сказать, что операции отношения, кроме, естественно, операций "равно" и "не равно", как бы раздваиваются: есть "знаковое больше" и "беззнаковое больше" и т.д. Это учитывается в различных мнемониках этих команд.

В Таблице 6.2 приведены мнемоники команд условного перехода. Некоторые команды имеют разную мнемонику, но выполняются одинаково (переводятся компилятором Ассемблера в одинаковые машинные команды), такие команды указаны в одной строке этой таблицы.

Таблица 6.2. Мнемоника команд условного перехода

КОП	Условие перехода	
	Логическое условие перехода	Результат (<code>rez</code>) команды вычитания или соотношение операндов <code>op1</code> и <code>op2</code> команды сравнения
je jz	<code>ZF = 1</code>	<code>rez = 0</code> или <code>op1 = op2</code> Результат = 0, операнды равны
jne jnz	<code>ZF = 0</code>	<code>rez <> 0</code> или <code>op1 <> op2</code> Результат <> 0, операнды не равны
jg jnle	<code>(SF=OF) and (ZF=0)</code>	<code>rez > 0</code> или <code>op1 > op2</code> Знаковый результат > 0, op1 больше op2
jge jnl	<code>SF = OF</code>	<code>rez >= 0</code> или <code>op1 >= op2</code> Знаковый результат >= 0, т.е. op1 больше или равен (не меньше) op2
jl jnge	<code>SF <> OF</code>	<code>rez < 0</code> или <code>op1 < op2</code> Знаковый результат < 0, т.е. op1 меньше (не больше или равен) op2
jle jng	<code>(SF<>OF) or (ZF=1)</code>	<code>rez <= 0</code> или <code>op1 <= op2</code> Знаковый результат <= 0, т.е. op1 меньше или равен (не больше) op2
ja jnb	<code>(CF=0) and (ZF=0)</code>	<code>rez > 0</code> или <code>op1 > op2</code> Беззнаковый результат > 0, т.е. op1 выше (не ниже или равен) op2
jae jnb jnc	<code>CF = 0</code>	<code>rez >= 0</code> или <code>op1 >= op2</code> Беззнаковый результат >= 0, т.е. op1 выше или равен (не ниже) op2
jb jnae	<code>CF = 1</code>	<code>rez < 0</code> или <code>op1 < op2</code> Беззнаковый результат < 0, т.е.

¹ Определённые команды условных переходов (**loop**, **jesxz** и некоторых другие), к сожалению, допускают только операнд формата `i8`, это будет подробно описано далее.

jc		op1 ниже (не выше или равен) op2
jbe jna	(CF=1) or (ZF=1)	rez <= 0 или op1 <= op2 Беззнаковый результат <= 0, т.е. op1 ниже или равен (не выше) op2
js	SF = 1	Знаковый бит результата (7-й, 15-й или 31-ый) равен единице
jns	SF = 0	Знаковый бит результата (7-й, 15-й или 31-ый) равен нулю
jo	OF = 1	Флаг переполнения равен единице
jno	OF = 0	Флаг переполнения равен нулю
jp jpe	PF = 1	Флаг чётности ¹ равен единице
jnp jpo	PF = 0	Флаг чётности равен нулю
jcxz jecxz jrcxz	CX = 0 ECX = 0 RCX = 0	Значение регистра CX/ECX равно нулю RCX равно нулю (в 64-битных ЭВМ)

Как видно из приведённой таблицы, команд условного перехода достаточно много, поэтому понятно, почему для *младших* моделей семейства реализован только один формат – *короткий* относительный прямой переход. На старших моделях, наряду с коротким, реализован и *длинный* (формата i32) условный переход, тип перехода выбирается компилятором Ассемблера MASM автоматически, в зависимости от расстояния от команды перехода до нужной метки. Отметим, что несмотря на многочисленность команд условного перехода, для кодирования условия перехода в командах используется всего четыре бита (если не принимать в расчёт команду условного перехода по регистру ECX).

В качестве примера использования команд условного перехода рассмотрим программу, которая вводит знаковое число A и вычисляет значение X по формуле

$$X := \begin{cases} (A+1) * (A-1), & \text{при } A > 10000 \\ 4, & \text{при } A = 10000 \\ (A+1) \bmod 7, & \text{при } A < 10000 \end{cases}$$

На языке Free Pascal решение этой задачи можно записать в виде

```
var A,X: Longint;
begin
  Read(A);
  if A>10000 then X:=(A+1)*(A-1) else
  if A<10000 then X:=(A+1) mod 7
  else X:=4;
  Writeln(X)
end.
```

Очевидно, что при выполнении этой программы могут произойти различные аварийные ситуации. Например, если величины (A+1) или (A+1) * (A-1) выйдут за допустимый диапазон, то в режиме {**\$R-**} будет выдан неверный ответ, а в режиме {**\$R+**} будет аварийное завершение программы. Попробуем в программе на Ассемблере всегда выдавать или правильный ответ, или диагностику об ошибке (с возможностью продолжить выполнение программы).

```
include iconsole.inc
.data
  A dd ?
  X dd ?
.code
start:
  inint A;          Read(A)
  mov  eax,A;       eax := A
  mov  ebx,eax;     ebx := A
  inc  eax;         eax := A+1
  jo   Err;         A+1 вне допустимого диапазона
  cmp  ebx,10000;   Сравнение A и 10000
```

¹ Флаг чётности равен единице, если в восьми младших битах результата содержится *чётное* число двоичных единиц, этот флаг используется редко.

```

    jle L1;          Вниз - по первой ветви вычисления X
    dec ebx;         ebx := A-1, всегда хорошо, т.к. X>10000
    imul ebx;        <edx, eax>:= (A+1) * (A-1)
    jo Err;          (A+1) * (A-1) не помещается в eax
    mov X, eax;       Результат берётся только из eax
L: outintln X;       Вывод результата
    exit
L1: j1 L2;           Вниз - по второй ветви вычисления X
    mov X, 4
    jmp L;           На вывод результата
L2: mov ebx, 7;      Третья ветвь вычисления X
    cdq;             <edx, eax> := int64 (A+1)
    idiv ebx;        edx:=<edx, eax> mod 7; eax:=<edx, eax> div 7
    mov X, edx
    jmp L;           На вывод результата
Err:
    outstrln 'Ошибка - плохое значение A!'
    exit;            Можно было бы продолжить программу
end start

```

В этой программе сначала кодировалось вычисление по первой ветви алгоритма, затем по второй и, наконец, по третьей. Программист, однако, может выбрать и другую последовательность кодирования ветвей, это не влияет на суть дела. Далее, предусматривается выдача аварийной диагностики (для простоты одинаковой), если результаты операций сложения $(A+1)$ или произведения $(A+1) * (A-1)$ слишком велики и не помещаются в переменную-результат X.

Для увеличения и уменьшения операнда на единицу использовались команды

```
inc op1    и    dec op1
```

Напомним, что, например, команда `inc eax` эквивалентна команде `add eax, 1`, но *не меняет* флаг CF. Таким образом, после этих команд нельзя проверить флаг переноса, чтобы определить, правильно ли выполнились такие операции над *беззнаковыми числами*, это необходимо учитывать в надёжном программировании. Здесь, однако, числа *знаковые*, поэтому всё в порядке.

Обратите также внимание, что использовалась команда *сверхдлинного* деления, попытка использовать здесь длинное деление, например

```

L2: mov bx, 7;      Третья ветвь вычисления X
    cwd;           <dx, ax> := Longint (A+1)
    idiv bx;       dx:=<dx, ax> mod 7; ax:=<dx, ax> div 7

```

часто приводит к ошибке. Здесь остаток от деления $(A+1)$ на число 7 всегда поместится в регистр DX, однако *частное* $(A+1) \text{ div } 7$ часто может *не поместиться* в регистр AX (пусть, например, $A=700000$, тогда $(A+1) \text{ div } 7 = 100000$ – не поместится в регистр AX).

В языке Ассемблера MASM, начиная с версий 6.xx, введена директива `.if` для облегчения программирования условных операторов. Директива реализована в виде стандартной макрокоманды Ассемблера, синтаксис этой директивы:

```

.if <логическое выражение>
    <предложения Ассемблера>
[.elseif <логическое выражение>
    <предложения Ассемблера>]
[.else
    <предложения Ассемблера>]
.endif

```

Необязательные части заключены в квадратные скобки. Логические выражения вычисляются во время счёта программы, в них можно использовать операции отношений `==` (равно), `!=` (не равно), `>` (больше), `<` (меньше), `<=` (меньше или равно) и `>=` (больше или равно). Логические операции задаются в виде `&&` (**and**), `||` (**or**) и `!` (**not**). Для проверки флагов используются конструкции **CARRY?**, **OVERFLOW?**, **SIGN?** и **ZERO?**. Например, конструкция

```

X    dd ?
    . . .
.if eax<X && !CARRY?
    add eax, 10

```



```
.else
    add ebx, eax
.endif
```

на Паскале эквивалентна условному оператору

```
if (eax<X) and not CF
then  eax:=eax+10
else  ebx:=ebx+eax
```

и будет преобразована компилятором Ассемблера в такой набор команд:

```
cmp  eax, X
jae  @C0001; на ветвь else
jc   @C0001; на ветвь else
add  eax, 10
jmp  @C0002
@C0001:
add  ebx, eax
@C0002:
```

Как видно, компилятор Ассемблера использовал условный переход **jae**, т.е. считает значение регистра EAX и переменной X *беззнаковыми*. Так происходит, когда компилятор Ассемблера не видит в логическом выражении *явного* знака минус, и в этом выражении не используются переменные, описанные при помощи *знаковых* директив резервирования памяти **sbyte**, **sword** или **sdword**, а иначе он будет считать значения *знаковыми*, например:

```
.if  eax<1; тогда  cmp  eax, 1
                        jae  @C0001

.if  eax<-1; тогда  cmp  eax, -1
                        jge  @C0001

X  sdword ?
. . .
.if  eax<X; тогда  cmp  eax, X
                        jge  @C0001
```

По таким же правилам различаются знаковые и беззнаковые величины и при вычислении логических выражений в условных макрооператорах, о чем будет рассказано в главе, посвященной макро средствам языка Ассемблер.

Начиная с процессора Intel 486 появилась команда *условной пересылки* **cmpxchg** (compare and swap – сравнение с обменом), имеющая один неявный и два явных операнда. Синтаксис команды:

```
cmpxchg  op1, op2
```

Для этой команды существуют следующие допустимые форматы неявного, первого и второго операндов:

Неявный операнд	op1	op2
AL	r8, m8	r8
AX	r16, m16	r16
EAX	r32, m32	r32
RAX	r64, m64	r64

Сначала сравнивается неявный операнд с первым операндом и устанавливаются флаги, затем, если ZF=1 (т.е. неявный операнд равен первому операнду), то op1:=op2, иначе (неявный операнд не равен первому операнду) <неявный операнд>:=op1. Команда является неделимой операцией, т.е. при её выполнении, как и для команды **xchg**, не снимается блокировка с шины обмена данными с оперативной памятью.

Начиная с процессора Intel P5 появилась команда *условного обмена* **cmpxchg8b** (compare and swap 8 byte – сравнение с обменом 8 байт), имеющая два неявных операнда, это регистровые пары <EDX, EAX> и <ECX, EBX>, а также один явный операнд формата m64. Синтаксис команды:

```
cmpxchg8b  op1; op1=m64
```

Сначала команда сравнивает 64-разрядное число <EDX, EAX> с op1 и устанавливает флаги. Если операнды равны (т.е. ZF=1), то выполняется присваивание op1:=<ECX, EBX>, иначе выполняется присваивание <EDX, EAX>:=op1. Видно, что эта длинная команда, она выполняется за целых 10

процессорных тактов. С использованием префикса **Lock** эта единственная команда, которая является неделимой операцией, позволяющей для 32-битных машин записывать в память сразу 8 байт.

Начиная с процессора Pentium Pro можно также использовать новые команды *условной пересылки* **CMOVcc**:¹

CMOVcc op1, op2

где после **CMOV** указывается мнемоника любой команды условного перехода, например, **cmovg**, **cmovae**, **cmovz** и т.д. Допустимые форматы операндов такие же, как и в команде пересылки **mov**. Команда сначала проверяет условие и, если оно истинно, выполняет пересылку op1:=op2. Например, следующий оператор для беззнаковых величин:

```
; if eax<10 then ebx:=eax else ecx:=10
    cmp    eax,10
    jae    L2
    mov    ebx,eax
    jmp    L3
L2: mov    ecx,10
L3:
```

с помощью этих новых команд можно переписать без использования команд условного перехода и без меток в виде:

```
cmp    eax,10
cmovb  ebx,eax
cmovae ecx,10
```

С командами условного перехода тесно связаны и так называемые команды установки бита

SETcc op1

где после **SET** указывается мнемоника любой команды условного перехода, например, **setge**, **setb**, **setz** и т.д. Команда записывает в свой операнд формата r8 или m8 значение единица, если заданное условие выполнено и ноль в противном случае. По-существу, так можно запомнить в переменной размером в байт "на будущее" выработанное условие для последующих команд перехода.ⁱⁱ

6.7.3. Команды цикла

1. Начальник всегда прав.

2. Если начальник не прав, см. Пункт 1.

Цикл с постусловием

Для организации циклов на Ассемблере можно использовать команды условного перехода. Например, цикл языка Паскаль с предусловием `while X<0 do S` для целой *знаковой* переменной X можно реализовать в виде следующего фрагмента на Ассемблере:

```
jmp    L2
L1:
; Здесь оператор S
L2: cmp    X,0; Сравнить X с нулём
    jl     L1
```

В языке Ассемблера, начиная с версий 6.xx, введена директива **.while** для облегчения программирования цикла с предусловием. Директива реализована в виде стандартной макрокоманды Ассемблера, синтаксис этой директивы:

```
.while <логическое выражение>
    <предложения Ассемблера>
.endw
```

Например, конструкция

```
.while eax>0 || ZERO?; (eax>0) or (ZF=1)
    sub    eax,1
.endw
```

будет преобразована компилятором Ассемблера в такой набор команд:

```
jmp    @C0001
@C0002:
    sub    eax,1
@C0001:
```

¹ Аналогичные команды **FCMOVcc** предусмотрены и для вещественных чисел.

```

cmp eax, 0
ja @C0002; беззнаковое eax>0
jz @C0002; ZF=1

```

Оператор цикла с постусловием `repeat S1; S2; ... Sk until X<0` можно реализовать в виде такого фрагмента на Ассемблере:

```

L:      ; S1
        ; S2
        . . .
        ; Sk
cmp X, 0; Сравнить X с нулём
jge L

```

В языке Ассемблера, начиная с версий 6.xx, введена директива `.repeat` для облегчения программирования цикла с постусловием. Директива реализована в виде стандартной макрокоманды Ассемблера, синтаксис этой директивы:

```

.repeat
    <предложения Ассемблера>
.until <логическое выражение>

```

Например, конструкция

```

.repeat
    sub eax, 1
.until eax==0 || !CF?; (eax=0) or (CF=0)

```

будет преобразована компилятором Ассемблера в такой набор команд:

```

@C0001:
    sub eax, 1
    je @C0002; if eax=0 then goto @C0002
    jnc @C0002; if CF=0 then goto @C0002
    jmp @C0001
@C0002:

```

Внутри циклов `.whileendw` и `.repeatuntil` можно также использовать условную или безусловную директиву досрочного выхода из цикла

```
.break [ .if <логическое выражение> ]
```

и директиву

```
.continue [ .if <логическое выражение> ]
```

которая производит условный или безусловный переход на проверку логического условия продолжения выполнения циклов `.whileendw` и `.repeatuntil`. Директивы `.if`, `.while`, `.repeat`, `.break` и `.continue` в примерах программ этой книги использоваться не будут.

Заметьте, что цикл с постусловием требует для своей реализации на одну команду (и на одну метку) меньше, чем цикл с предусловием, а значит их использование предпочтительнее.¹

В том случае, если число повторений тела цикла известно до начала исполнения этого цикла, то в языках высокого уровня (например, в Паскале) наиболее целесообразно использовать *цикл с параметром*. Для организации цикла с параметром в Ассемблере можно использовать специальные *команды цикла*. Команды цикла, по сути, тоже являются командами условного перехода, однако они реализуют только *короткий прямой относительный* переход. Команда цикла

```
loop L; Метка L заменится на операнд i8
```

использует неявный операнд – регистр ECX и её выполнение может быть так описано на Паскале:

```

Dec (ECX); {Это часть команды loop, т.е. флаги не меняются!}
if ECX<>0 then goto L;

```

Как видим, регистр ECX (который так и называется (расширенным) регистром-счётчиком цикла – extended loop counter), используется этой командой именно как параметр цикла.¹ Лучше всего эта команда цикла подходит для реализации цикла с параметром языка Паскаль вида

¹ Например, компиляторы с языка C, к удивлению адептов этого языка, реализуют их любимые циклы `while` в виде "чистых" циклов `repeat`, если уверены, что тело цикла должно выполниться хотя бы один раз, в противном случае имитируя цикл `repeat`, первой же командой передавая управление на проверку выхода из цикла. Кроме того, процессоры фирмы Intel оптимизированы под эффективное выполнение именно циклов с постусловием (`repeat` и `loop`), т.е. с передачей в цикле управления *назад* по тексту программы.

```
for ECX:=N downto 1 do S
```

Этот оператор можно реализовать таким фрагментом на Ассемблере:

```
mov    ECX,N
jecxz  L1
L: . . .; Тело цикла -
. . .; оператор S
loop  L
L1: . . .
```

Обратите внимание: так как цикл с параметром языка Паскаль по существу является циклом с предусловием, то до начала его выполнения проверяется исчерпание значений для параметра цикла с помощью команды условного перехода `jecxz L1`, которая именно для этого и была введена в язык машины. Ещё раз напоминаем, что команды циклов, как и все команды переходов, *не меняют* флагов. К сожалению, команда `jecxz` реализует короткий переход, если расстояние до метки L1 велико, следует использовать пару команд `test ecx,ecx` и `jz L1`, но флаги здесь *меняются*.

Описанная выше команда цикла выполняет тело цикла ровно N раз, где N – *беззнаковое* число, занесённое в регистр-счётчик цикла ECX перед началом цикла. Особым является случай, когда N=0, в этом случае цикл выполнится 2^{32} раз. К сожалению, никакой другой регистр *нельзя* использовать как счётчик для организации этого цикла (т.к. это неявный параметр команды цикла). Кроме того, в приведённом выше примере реализации цикла тело этого цикла не может быть слишком большим, иначе команда `loop L` *не сможет* передать управление на метку L. Таким образом, Во всех моделях этой архитектуры для реализации цикла `loop` применяется короткий относительный переход, как следствие тело цикла не может быть слишком большим (это 128 байт, т.е. порядка 30–40 машинных команд).²

Архитектурное решение о том, что циклы и условные переходы лучше реализовать командами *короткого* перехода, опирается на так называемое свойство **локальности** программ. Для *команд* программы это свойство означает, что большую часть времени процессор выполняет команды, расположенные (локализованные) внутри относительно небольших участков программы (это и есть программные циклы), и достаточно редко переходит из одного такого участка в другой. Аналогичным свойством обладают и обрабатываемые в программе *данные*. Это называется *пространственной* локальностью команд и данных. Для команд может существовать и *временная* локальность, когда в цикле выполняются команды, не обязательно располагающиеся рядом в памяти (например, в цикле вызываются две процедуры). Как будет рассказано позже, на свойстве локальности основано существование в современных компьютерах специальной памяти типа кэш, а также так называемого *расслоения* оперативной памяти.

В качестве примера использования команды цикла решим следующую задачу. Требуется ввести *беззнаковое* число $N \leq 500$, затем ввести N *знаковых* целых чисел и вывести сумму тех из них, которые принадлежат диапазону $-2000..5000$. На языке Free Pascal эта программа могла бы выглядеть следующим образом (программа заранее записана так, чтобы было легче перенести её на Ассемблер):

```
const MaxN=500;
var N: longint; eax,ecx: longint; S: longint=0;
begin { $R+ }
  Write('Введите N<=500 : ');
  Read(N);
```

¹ Используя команду-префикс смены длины регистра с кодом операции 66h можно использовать в качестве счётчика цикла 16-битный регистр CX, но никакого смысла это не имеет. Для 64-битных ЭВМ можно использовать также счетчик цикла в регистре RCX.

² На современных процессорах команда `loop L` проигрывает "прямому" программированию, она выполняется *медленнее*, чем пара команд `dec ecx` и `jnz L`. По так называемой технологии Macro-Fusion, реализованной на современных процессорах, некоторые пары коротких команд могут "спариваться", они сливаются в одну микрооперацию и выполняются сразу на двух конвейерах за один такт работы процессора. Кроме того, команда `loop L` реализует только *короткий* (i8) условный переход, в то время как команда `jnz L` может выполнять как короткий, так и длинный (i32) условный переход, к тому же, для организации цикла можно использовать не только регистр `ecx`. Исходя из вышесказанного сейчас команда `loop` не используется компиляторами с языков высокого уровня. Единственным преимуществом команды `loop L` является то, что она не портит флаги и немного короче.

```

    if N > MaxN then Writeln('Ошибка - большое N!') else
    if N > 0 then begin
        Write('Вводите целые числа');
        for ecx:=N downto 1 do begin Read(eax);
            if (-2000<=eax) and (eax<=5000) then S:=S+eax
        end
    end;
    Writeln('S=',S)
end.

```

На Ассемблере можно предложить следующее решение этой задачи.

```

include console.inc
MaxN equ 500; Аналог const MaxN=500; Паскаля
.data
    N    dd    ?
    S    dd    0; Начальное значение = 0
.code
start:
    inintln    N, 'Введите N<=500 '
    cmp        N, MaxN
    jbe        L1
    outstrln    "Ошибка - большое N!"
    exit
L1:mov        ecx, N; Счётчик цикла
    cmp        ecx, 0
    je         Pech; На печать результата при N=0
    outstrln    'Вводите целые числа : '
L2:inint      eax; Ввод очередного числа
    cmp        eax, -2000
    jl         L3
    cmp        eax, 5000
    jg         L3; Проверка диапазона
    add        S, eax; Суммирование
    jo         Error; Проверка на переполнение S
L3:dec        ecx
    jnz        L2; Цикл
Pech:
    outintln    S, , 'S='
    exit
Error:
    outstrln    "Ошибка - большая сумма!"
    exit
end start

```

Как видно, так как в программе на Паскале задана директива `{ $R+ }`, то и приведённая программа на Ассемблере также проверяет корректность полученного результата. Максимальное количество вводимых чисел задано с использованием директивы эквивалентности `MaxN equ 500`, эта директива определяет *числовую макроконстанту* MaxN. Директива эквивалентности есть указание компилятору Ассемблера о том, что всюду далее в программе, где встретится имя MaxN, надо подставить вместо него операнд этой директивы, т.е. в нашем случае число 500 (почему MaxN называется *макроконстантой* станет ясно при описании макросредств). Таким образом, это почти полный аналог описания константы в языке Паскаль.¹

¹ Если не принимать во внимание то, что константа в Паскале имеет *тип*, это позволяет контролировать её использование в программе, а директива эквивалентности в Ассемблере – это просто указание о *текстовой подстановке* вместо имени заданного операнда директивы эквивалентности. Кроме числовых макроконстант директива `equ` может задавать и *текстовые макропеременные*, в этом случае параметр директивы `equ` должна быть текстовая строка, сбалансированная по скобкам и кавычкам, она не должна содержать вне скобок и кавычек символ ";" (т.к. это начало комментария). В параметре директивы `equ` можно использовать имена, описанные как до, так и после этой директивы. Иногда перед подстановкой параметра на место макроконстанты и макропеременной производится также некоторое *вычисление* операнда директивы эквивалентности, подробнее об этом можно прочитать в учебниках [5-7].

В качестве ещё одного примера рассмотрим использование циклов при обработке массивов. Пусть необходимо составить программу для решения следующей задачи. Задана константа $N=200000$, надо ввести массивы X и Y по N *беззнаковых* чисел в каждом массиве и вычислить выражение

$$S := \sum_{i=1}^N X[i] * Y[N-i+1]$$

Так как числа беззнаковые, то выход каждого из произведений и первой же частичной суммы за диапазон, допустимый для формата **dd**, должен вызывать аварийную диагностику и прекращение выполнения программы. Ниже приведена программа на Ассемблере, решающая эту задачу.

```
include console.inc
N equ 200000; Аналог Const N=200000; Паскаля
.data
S dd 0; искомая сумма
Prig db "Вводите числа массива ",0
.data?
X dd N dup (?); Массив X длины 4*N байт
Y dd N dup (?); Массив Y длины 4*N байт
.code
begin_of_my_good_program:
    outstr offset Prig; приглашение к вводу
    outchar 'X';           массива X
    newline
    mov     ecx,N;   счётчик цикла
    mov     ebx,0;   индекс массива
L1:inint    X[ebx]; ввод очередного X[i]
    add     ebx,4;   это i:=i+1
    dec     ecx
    jnz     L1
    outstr  offset Prig; приглашение к вводу
    outchar 'Y';           массива Y
    newline
    mov     ecx,N;   счётчик цикла
    mov     ebx,0;   индекс массива
L2:inint    Y[4*ebx]; на один байт длиннее, чем Y[ebx] !
    inc     ebx;     это i:=i+1
    dec     ecx
    jnz     L2;      Ввод второго массива
    mov     ebx,offset X; ebx:=адрес X[1]
    mov     esi,offset Y+4*N-4; esi:=адрес Y[N]
    mov     ecx,N;   счётчик цикла
L3:mov      eax,[ebx]; первый сомножитель X[i]
    mul     dword ptr [esi]; умножение на Y[N-i+1]
    jc      Err;     большое произведение
    add     S,eax;    S:=S+X[i]*Y[N-i+1]
    jc      Err;     большая сумма
    add     ebx,type X; это ebx:=ebx+4, т.е. i:=i+1
    sub     esi,4;     это esi:=esi-4, т.е. i:=i-1
    dec     ecx
    jnz     L3;      цикл суммирования
    outwordln S,,"Сумма = "
    exit
Err:
    outstr "Ошибка - большое значение!"
    exit
end        begin_of_my_good_program
```

Подробно прокомментируем эту программу. Так как длина каждого элемента массива равна четырём байтам, то под каждый из массивов соответствующая директива зарезервирует $4*N$ байт памяти. При вводе массивов использован индексный регистр `ebx`, в котором находится индекс текущего элемента, т.е. его *смещение* от начала массива.

В цикле суммирования произведений для доступа к элементам массивов использован другой приём, чем при вводе – регистры-указатели EBX и ESI, в этих регистрах находятся *адреса* очередных элементов массивов. Напомним, что адрес – это *смещение* элемента относительно *начала сегмента памяти* (в отличие от индекса элемента – это смещение от *начала массива*). Использование адресов элементов вместо их индексов позволяет работать с элементами массива, не зная его *имени*. Это будет очень важно при программировании процедур и функций на Ассемблере.

При записи команды умножения

```
mul dword ptr [esi]; X[i]·Y[N-i+1]
```

необходимо *явно* задать размер второго сомножителя с помощью оператора явной смены типа **dword ptr**, он *приказывает* считать, что операнд [ESI] является адресом двойного слова.¹ Это необходимо, так как по виду операнда [ESI] компилятор Ассемблера не может сам "догадаться", что это элемент массива *размером в двойное слово*. В качестве альтернативы можно было бы использовать команду, где есть имя Y, явно задающее размер операнда (однако эта команда на 5 байт длиннее!)

```
mul Y[4*ecx-4]; X[i]·Y[N-i+1]
```

В команде

```
add ebx, type X; это ebx:=ebx+4
```

для задания размера элемента массива использован оператор **type**. Параметром этого оператора обычно является имя из программы, значением оператора **type <ИМЯ>** является целое число длиной два байта – *тип* данного имени. Для имён областей памяти значение типа – это длина этой области в байтах (учтите, что для массива это почти всегда длина одного элемента, а не всего массива!). Для меток команд и имен процедур это отрицательное число –252 (это значение служебной константы **near**), для меток и процедур из других модулей это число –250 (значение служебной константы **far**). Имена регистров r8, r16 и r32 имеют тип 1, 2 и 4 соответственно. Имена, описанные в директивах эквивалентности константным выражениям (например, **N equ 10**), имеют тип ноль, это же число выдаётся при попытках использовать оператор **type** без имени (например, **type [ebx]** или **type 333**). Применение оператора **type** к служебным и неописанным именам приводит к ошибкам компиляции, например:

```
mov eax, type add; Error A2008: syntax error
T equ ABC
mov eax, type T; Error A2006: undefined symbol: ABC
```

Вы, наверное, уже почувствовали, что программирование на Ассемблере сильно отличается от программирования на языке высокого уровня (например, на Паскале). Чтобы подчеркнуть это различие, рассмотрим пример задачи, связанной с обработкой матрицы, и решим её на Паскале и на Ассемблере.

Пусть дана матрица целых чисел и надо найти сумму элементов, которые расположены в строках, начинающихся с отрицательного значения. Для решения этой задачи на Паскале можно предложить следующий фрагмент программы (для простоты ввод не реализован, а указан комментарием)

```
Const N=20; M=30;
Var
  X: array[1..N,1..M] of integer;
  i,j: integer; Sum: longint;
Begin
  { Ввод матрицы X }
  Sum:=0;
  for i:=1 to N do
    if X[i,1]<0 then
      for j:=1 to M do Sum:=Sum+X[i,j];
```

Для переноса программы с языка высокого уровня (в нашем случае с Паскаля) на язык низкого уровня (Ассемблер) необходимо выполнить два преобразования, которые в программистской литературе часто называются *отображениями*. Как говорят, надо отобразить с языка высокого уровня на язык низкого уровня структуру данных и структуру управления программы. В нашем примере глав-

¹ Оператор **ptr** является аналогом явного преобразования типа в языках высокого уровня. Он преобразует тип операнда команды в памяти (но не на регистре), перед **ptr** можно указывать любой тип (**byte**, **word**, **dword**, тип структуры и т.д.).

ная структура данных в программе на Паскале – это прямоугольная таблица (матрица) целых чисел. При отображении матрицы на линейную память секции данных в Ассемблере приходится, как говорят, *линеаризовать* матрицу. Этим мудрёным термином обозначают совсем простой процесс: сначала в секции данных размещается первая строка матрицы, сразу вслед на ней – вторая, и т.д. Для нашего примера в некоторой секции надо каким-то образом зарезервировать $2 * N * M$ байт памяти (т.к. элемент матрицы занимает два байта). Процесс линеаризации усложняется, когда надо отобразить массивы больших размерностей (например, четырёхмерный массив – совсем обычная вещь в задачах из области физики твёрдого тела или механики сплошной среды). Подумайте, как надо сделать отображение такого массива на линейную память. Кроме того, надо сказать, что некоторые языки программирования высокого уровня требуют другого способа линеаризации. Например, для языка Фортран нужно сначала разместить в памяти секции первый столбец матрицы, потом второй столбец и т.д.

Теперь займёмся отображением структуры управления нашей программы на Паскале (а попросту говоря – её условных операторов и циклов) на язык Ассемблера. Сначала обратим внимание на то, что переменные *i* и *j* несут в программе на Паскале двойную нагрузку: это одновременно и счётчики циклов, и индексы элементов массива. Такое совмещение функций упрощает понимание программы и делает её очень компактной по внешнему виду, но не проходит даром. Теперь, чтобы по индексам элемента массива вычислить его адрес в памяти, приходится выполнить достаточно сложные действия. Например, адрес элемента $X[i, j]$ компилятору с Паскаля приходится вычислять так:

$$\text{Адрес}(X[i, j]) = \text{Адрес}(X[1, 1]) + 2 * M * (i - 1) + 2 * (j - 1)$$

Эту формулу легко понять, учитывая, что для Ассемблера матрица хранится в памяти компьютера по строкам (сначала первая строка, затем вторая и т.д.), и каждая строка имеет длину $2 * M$ байт. Буквальное вычисление адресов элементов по приведённой выше формуле (а именно так чаще всего и делает Паскаль-машина) приводит к весьма неэффективной программе. При отображении этих циклов на язык Ассемблера лучше всего **разделить** функции счётчика цикла и индекса элементов. В качестве счётчика будем использовать регистр ECX (он и специализирован для этой цели), а адреса элементов матрицы лучше хранить в индексных регистрах (EBX, ESI, EDI и т.д.). Исходя из этих соображений, можно так переписать фрагмент на Паскале, предвидя будущий перенос на Ассемблер.

```
const N=20; M=30;
var   X: array[1..N,1..M] of integer;
      Sum: longint=0;
      ecx,oldecx: longword; ebx: ^integer;
begin
  { Ввод матрицы X }
  ebx:=^X[1,1]; {Так в Паскале нельзя}1
  for ecx:=N downto 1 do
    if ebx^<0 then begin
      oldcx:=ecx;
      for ecx:=M downto 1 do begin
        Sum:=Sum+ebx^; ebx:=ebx+2 {Так в Паскале нельзя}
      end;
      ecx:=oldcx
    end
    else ebx:=ebx+2*M {Так в Паскале нельзя}
```

Теперь осталось переписать этот фрагмент программы на Ассемблере:

```
N      equ 20
M      equ 30;  Числовая макроконстанта
oldecx equ edi; Текстовая макропеременная
.data
  X    dw N*M dup (?)
  Sum  dd 0
.code
; Здесь ввод матрицы X
  mov  ebx,offset X; ebx:=^X[1,1]
```

¹ В языке Free Pascal вместо недопустимого в стандарте Паскаля оператора $ebx := ^X[1, 1]$; можно использовать оператор присваивания $ebx := @X[1, 1]$.


```

    mov     ecx, N
L1: cmp     word ptr [ebx], 0
    jge     L3
    mov     oldecx, ecx
    mov     ecx, M
L2: mov     ax, [ebx]
    cwde;   eax:=Longint(ax)
    add     Sum, eax
    add     ebx, 2; i:=i+1
    dec     ecx
    jnz     L2
    mov     ecx, oldecx
    jmp     L4
L3: add     ebx, 2*M; На след. строку
L4: dec     ecx
    jnz     L1

```

Приведённый пример очень хорошо иллюстрирует стиль мышления программиста на Ассемблере. Для доступа к элементам обрабатываемых данных применяются указатели (ссылочные переменные, значениями которых являются адреса) на регистрах, и используются операции над этими адресами (так называемая *адресная арифметика*). Получающиеся программы могут максимально эффективно учитывать все особенности как решаемой задачи, так и архитектуры используемого компьютера. Заметим, что применение адресов и адресной арифметики свойственно и некоторым языкам высокого уровня (например, языку C), которые ориентированы на использование особенности машинной архитектуры для написания более эффективных программ.

Вернёмся теперь к описанию команд цикла. В языке Ассемблера есть и другие команды цикла, которые могут производить досрочный (до исчерпания счётчика цикла) выход из цикла с параметром. Как и для команд условного перехода, для мнемонических имен некоторых из них существуют синонимы, которые будут разделяться в описании этих команд символом / (слэш).

Команда

loopz/loope L

выполняется по схеме

Dec (ECX) ; **if** (ECX<>0) **and** (ZF=1) **then goto L;**

А команда

loopnz/loopne L

выполняется по схеме

Dec (ECX) ; **if** (ECX<>0) **and** (ZF=0) **then goto L;**

В этих командах необходимо учитывать, что операция Dec (ECX) является частью команды цикла и *не меняет* флага ZF. Как видим, досрочный выход из таких циклов может произойти при соответствующих значениях флага нуля ZF. Такие команды используются в основном при работе с массивами, для полного усвоения этого материала Вам необходимо изучить какой-нибудь учебник по Ассемблеру [5-7].

Вопросы и упражнения

1. Когда у программиста может появиться необходимость при написании своих программ использовать не более удобный для программирования язык высокого уровня, а перейти на язык низкого уровня (Ассемблер) ?
2. Можно ли выполнять команды из секции данных ?
3. В каких секциях могут располагаться области данных для хранения переменных ?
4. Почему реализованы две макрокоманды для вывода знаковых (**outint**) и беззнаковых (**outword**) чисел, в то время, как макрокоманда ввода целого числа всего одна (**inint**) ?
5. Почему в языке машины не реализована команда пересылки вида **mov CS, op2** ?
6. Какое преимущество имеет относительный переход перед абсолютным переходом ?
7. Для чего необходимы разные команды условного перехода после сравнения на больше или меньше знаковых и беззнаковых чисел ?

8. Что делать, если в команде цикла `loop L` необходимо передать управление на метку `L`, расположенную достаточно далеко от команды цикла ?
9. Чем отличается выполнение команды `inc op1` от выполнения команды `add op1, 1` ?
10. Почему для деления числа в формате слова на маленькие числа часто необходимо использовать команду длинного, а не короткого деления ?
11. Почему при реализации цикла на Ассемблере, если это возможно, надо выбирать цикл с постусловием, а не цикл с предусловием ?
12. Когда перед операндом формата `m16` необходимо ставить явное задание длины операнда в виде `word ptr` ?
13. Как работает оператор Ассемблера `type` ?

6.8. Работа со стеком

Кто думает, что постиг всё, тот ничего не знает.

Лао-цзы, VI век до н.э.

Прежде, чем двигаться дальше в описании команд перехода, необходимо изучить понятие аппаратного *стека* и рассмотреть команды машины для работы с этим стеком.¹ Стек в архитектуре компьютеров x86 называется область памяти, на текущую позицию стека в этой области указывает регистр ESP (Extended Stack Pointer), таким образом, стек в машине есть *всегда*. В архитектуре процессоров x86 стек хранится в памяти в перевёрнутом виде: при увеличении размера стека его вершина смещается в область памяти с меньшими адресами, а при чтении из стека – в область с большими адресами. В текущей позиции стека (регистре ESP) хранится смещение вершины стека от начала памяти сегмента стека (но, как уже говорилось, в плоской модели памяти все сегменты совпадают).

Стек есть аппаратная реализация абстрактной структуры данных *стек*, в машине предусмотрены специальные команды записи в стек и чтения из стека. В стек можно записывать (и, соответственно, читать из него) только машинные *слова* (**dw**) и *двойные слова* (**dd**), команды чтение и запись в стек байтов не предусмотрены. Это, конечно, не значит, что в стеке нельзя *хранить* байты, просто нет машинных *команд* для записи в стек и чтения из стека данных этого формата. Наличие стека не отменяет для нашего компьютера принципа Фон Неймана однородности памяти, поскольку одновременно стек является и просто областью оперативной памяти и, таким образом, возможен обмен данными с этой памятью с помощью обычных команд (например, команд пересылки **mov**).

В соответствии с определением стека как абстрактной структуры данных, последнее записанное в него слово будет храниться в вершине стека, и читаться из стека первым. Это так называемое правило "последний пришёл – первый вышел" (английское сокращение LIFO – Last In First Out). Вообще говоря, это же правило можно записать и как "первый пришёл – последний вышел" (английское сокращение FILO – First In Last Out). В литературе встречаются оба этих правила и их сокращения. В русскоязычной литературе стек иногда называют *магазином*, по аналогии с магазином автоматического оружия, в котором последний вставленный патрон выстреливается первым.

Будем изображать стек "растущим" снизу вверх, от текущей позиции к началу области памяти, занимаемой стеком. Как следствие получается, что конец стека фиксирован и будет расположен на рисунках снизу, а вершина движется вверх (при записи в стек) и вниз (при чтении из стека).

В любой момент времени регистр ESP указывает на вершину стека – это *последнее* записанное в стек слово или двойное слово. Таким образом, регистр ESP является специализированным регистром, следовательно, хотя на нём и можно выполнять арифметические операции сложения и вычитания, но делать это следует только тогда, когда необходимо изменить положение вершины стека. Стек будет изображаться, как показано на рис. 6.3.



Рис. 6.3. Так будет изображаться стек.

На этом рисунке, как и говорилось, стек растёт снизу вверх, занятая часть стека закрашена. В начале работы программы, когда стек пустой, регистр ESP указывает на первое слово за концом стека. [Здесь необходимо заметить, что в англоязычных книгах стек рисуют растущим сверху вниз, т.к. считается, что память надо изображать так, чтобы байты со старшими адресами находились сверху \(по английски High Addresses – "верхние" адреса\). Соответственно в дескрипторе сегмента есть специ-](#)

¹ Стек как структура данных определён в 1946 году Аланом Тьюрингом для хранения адресов возвратов из процедур, однако *запатентовали* эту идею в 1957 году немцы Клаус Самельсон и Фридрих Л.Бауэр.

альный флаг направления расширения сегмента ED (Expand Down), для сегмента стека можно установить ED=1, это говорит, что стек растёт вверх (а по английски Down, т.е. вниз). Флаг ED может использоваться для контроля переполнения стека, но для страничной организации памяти, используемой в плоской модели, контроль переполнения делается другими средствами, так что указанный флаг в стековом сегменте не используется. Именно поэтому и можно считать, что сегмент стека накладывается на остальные сегменты.

Программист может сам не описывать в своей программе секцию стека, она подставляется компилятором Ассемблера автоматически. Области памяти в стеке обычно не имеют имён, так как доступ к ним, как правило, производится только с использованием адресов на регистрах.¹

Обратим здесь внимание на одно важное обстоятельство. Перед началом работы со стеком необходимо загрузить в регистр ESP требуемое значение для пустого стека, поэтому перед началом выполнения программы этому регистру присвоит нужное значение специальная системная программа *загрузчик*. Загрузчик размещает программу в памяти и передаёт управление на команду, помеченную меткой, указанной в конце модуля в качестве параметра директивы **end**. Как видно, программа загрузчика выполняет, в частности, те же функции начальной установки значений необходимых регистров, которые в учебной трёхадресной машине выполняло устройство ввода при нажатии кнопки ПУСК.² При работе программы стек может расти (т.е. значение регистра ESP будет уменьшаться). При переполнении стека (выходе значения регистра ESP за верхнюю границу отведённой под стек области памяти), операционная система автоматически увеличивает размер стека (в пределах доступной памяти).ⁱⁱⁱ

Теперь приступим к изучению команд, которые работают со стеком (т.е. читают из него и пишут в него слова). Рассмотрим сначала те команды работы со стеком, которые *не являются* командами перехода. Команда

push op1

где единственный операнд op1 может иметь форматы r32, m32, i32, ES, DS, SS, FS, GS и CS записывает в стек двойное слово, определяемое своим операндом. Эта команда выполняется по правилу:

$ESP := (ESP - 4) \bmod 2^{32}$; $\langle ESP \rangle := op1$

Учтите, что 16-разрядные регистры ES, DS, SS, CS, FS и GS дополняются двумя байтами и записываются в стек виде двойного слова.

Особым случаем является команда

push ESP

Она выполняется по схеме

$\langle ESP - 4 \rangle := op1$; $ESP := (ESP - 4) \bmod 2^{32}$

Эти схемы различаются только тем, когда (до или после уменьшения значения ESP) само это значение записывается в стек.

Команда

pop op1

где op1 может иметь форматы r32, m32, ES, DS, SS, FS и GS читает из вершины стека двойное слово и записывает его в место памяти, определяемое своим операндом. Эта команда выполняется по правилу:

$op1 := \langle ESP \rangle$; $ESP := (ESP + 4) \bmod 2^{32}$

Для 16-битных регистров ES, DS, SS, FS и GS из прочитанного двойного слова в операнд записывается только одно (старшее) слово.

В принципе, операнд команд **push** и **pop** может иметь также форматы r16 и m16, команда с такими операндами записывает в стек или считывает из стека не 4, а только 2 байта. Пользоваться этими форматами не рекомендуется, потому что многие системные программы, в частности те, которые вызываются при выполнении макрокоманд, крайне "болезненно" относятся к случаю, когда значение регистра ESP не кратно 4, чаще всего они не работают. Исходя из этого, при программировании на

¹ Как задать переменным в стеке имена будет рассказано ниже, при описании процедур на Ассемблере.

² Это упрощённое изложение запуска программы на счёт, на самом деле загрузчик только передаёт программу на выполнение так называемому диспетчеру процессов, более подробно это будет описано в другой главе.

Ассемблере лучше придерживаться правила, по которому в стек пишутся и читаются из стека только двойные слова (**dd**).

Команды без явного параметра

pushf и **pushfd**

записывает в стек соответственно регистр флагов **FLAGS** и **EFLAGS**, а команды

popf и **popfd**

наоборот, читают из стека слово и записывает его в регистр флагов **FLAGS**, или читают двойное слово, и записывает его в регистр флагов **EFLAGS**.¹ Эти команды удобны для сохранения в стеке и восстановления значения регистра флагов.

В старших моделях семейства существуют удобные команды работы со стеком. Команды

pusha и **pushad**

последовательно записывает в стек регистры **AX**, **CX**, **DX**, **BX**, **SP** (этот регистр записывается *до* его изменения), **BP**, **SI** и **DI** или, соответственно, регистры **EAX**, **ECX**, **EDX**, **EBX**, **ESP** (этот регистр записывается *до* его изменения), **EBP**, **ESI** и **EDI**. И, наоборот, команды

popa и **popad**

последовательно считывает из стека и записывает значения в эти же регистры (но, естественно, в обратном порядке). Эти команды предназначены для сохранения в стеке и восстановления значений сразу всех регистров общего назначения.

В качестве примера использования стека рассмотрим программу для решения следующей задачи. Необходимо вводить целые *беззнаковые* числа до тех пор, пока не будет введено число ноль (признак конца ввода). Затем следует вывести *в обратном порядке* те из введённых чисел, которые принадлежат диапазону [2..100]. Ниже приведено возможное решение этой задачи.

```
include console.inc
.code
program_start:
    outstrln 'Вводите числа до нуля:'
    sub     ecx,ecx; хороший способ для ecx:=0
L:  inint   eax
    cmp     eax,0;   проверка конца ввода
    je      Pech;   на вывод результата
    cmp     eax,2
    jb      L
    cmp     eax,100
    ja      L;      проверка диапазона [2..100]
    push    eax;     запись "хорошего" числа в стек
    inc     ecx;     счетчик количества чисел в стеке
    jmp     L
Pech:
    outstrln 'Числа в обратном порядке:'
    cmp     ecx,0
    jz      Kon;     нет чисел в стеке
L1: pop     eax
    outword eax,10; Write(eax:10)
    dec     ecx
    jnz     L1
Kon:
    exit
end program_start
```

Заметим, что в этой программе нет собственно переменных, а только строковые константы, поэтому не описана отдельная секция данных, однако эта секция будет определена при обработке некоторых макрокоманд.

¹ При работе обычного пользователя в так называемом защищённом режиме некоторые биты регистра флагов (например, флаг запрета прерываний **IF** и поле **IOPL**, задающее уровень привилегий ввода/вывода) при чтении из стека в регистре **EFLAGS** *не меняются*.

Теперь, после того, как Вы научились работать с командами записи слов в стек и чтения слов из стека, вернемся к дальнейшему рассмотрению команд перехода.

6.9. Команды вызова процедуры и возврата из процедуры

*Мудр тот, кто знает не многое,
а нужное.*

Эсхил

Команды вызова процедуры по-другому называются командами *перехода с запоминанием точки возврата*,¹ что более правильно, так как их можно использовать и вообще без процедур. По своей сути это команды *безусловного перехода*, которые перед передачей управления в другое место программы запоминают в стеке адрес следующей за ними команды. Таким образом, обеспечивается возможность возврата в точку программы, следующую непосредственно за командой вызова процедуры. На языке Ассемблера эти команды имеют следующий вид:

call *op1*

Различают *близкий* (внутрисегментный) и *дальний* (межсегментный) вызовы. Для близкого вызова параметр *op1* может иметь следующие форматы: *i16*, *r16*, *m16*, *m32*, *r32* и *i32*. Как видно, по сравнению с командой обычного безусловного перехода **jmp** *op1* здесь не реализован близкий короткий относительный переход **call** *i8*, он бесполезен в практике программирования, так как почти всегда процедура находится достаточно далеко от точки её вызова. Форматы операнда *i16*, *r16* и *m16* практически не используются, они остались для совместимости с прежней 16-битной архитектурой процессора, для таких форматов компилятор Ассемблера вставляет впереди команду-префикс с кодом *66h*. Вызов близкой процедуры выполняется по следующей схеме:

Встек(EIP); **jmp** *op1*

Здесь запись **Встек(EIP)** обозначает действие "записать значение регистра EIP в стек". Как уже говорилось, при выполнении текущей команды, счётчик адреса EIP указывает на начало *следующей* команды. Заметим также, что отдельной команды **push** EIP в языке машины нет.

Рассмотрим теперь механизм возврата из процедур. Для возврата на команду программы, адрес которой находится на вершине стека, предназначена команда *возврата из процедуры*, по сути, это тоже команда безусловного перехода. Команда возврата из процедуры имеет следующий формат:

ret [*i16*]; Операнд на Ассемблере может быть опущен

На языке машины у этой команды есть две модификации, отличающиеся кодами операций: *близкий* (**retn**) возврат в пределах одного сегмента кода, и *дальний* (**retf**) возврат из процедуры, расположенной в "чужом" сегменте кода. В плоской модели памяти используется только один сегмент кода, поэтому применяется близкий возврат, дальний возврат делают системные процедуры, вызванные по команде **call far ptr**, такие процедуры описываются в разделе 6.11. Когда программист пишет команду **ret**, то компилятор Ассемблера сам пытается "сообразить", какую команду (**retn** или **retf**) надо поставить.

Команда *близкого* возврата из процедуры **retn** выполняется по схеме:

Изстека(EIP); ESP := (ESP + *i16*) **mod** 2³²

Здесь, по аналогии с командой вызова процедуры, запись **Изстека(EIP)** обозначает действие "считать из стека слово и записать его в регистр EIP", такой "отдельной" машинной команды нет.

Перейдём теперь к подробному рассмотрению программирования *близких* процедур на Ассемблере.

6.10. Программирование близких процедур на Ассемблере

*Если в вашей процедуре 10 параметров,
вероятно, вы что-то упускаете.*

Алан Перлис

¹ В некоторых книгах по Ассемблеру такие команды называют "командами перехода с возвратом", что, конечно, не совсем правильно, так как сами эти команды никакого "возврата" не производят.

В языке Ассемблера есть понятие процедуры – это участок программы между директивами **proc** и **endp**:

```
<имя процедуры> proc [,] [<спецификации процедуры>]
    тело процедуры
<имя процедуры> endp
```

Между этими двумя директивами располагается *тело* процедуры. Заметьте, что в самом машинном языке понятие процедуры отсутствует, а команда **call**, как уже говорилось, является просто командой безусловного перехода с запоминанием в стеке адреса следующей за ней команды. О том, что эта команда используется именно для вызова процедуры, знает только сам программист.

Первое предложение называется *заголовком* процедуры. Синтаксис Ассемблера допускает, чтобы тело процедуры было пустым, т.е. не содержало ни одного предложения. Как и в Паскале, допускается вложенность процедур одна в другую, однако это не дает никаких особых преимуществ и практически не используется. Заметьте также, что, по аналогии с Паскалем, имена *меток* локализованы в теле процедуры, т.е. они *не видны* извне, однако остальные имена (переменных, констант и т.д.), наоборот, видны из любого места модуля на Ассемблере. В том редком случае, когда на метку в теле процедуры необходимо перейти извне этой процедуры, эту метку надо сделать *глобальной* с помощью двух двоеточий, например:

```
MyProc proc
    Global_Label::
        add eax,ebx
MyProc endp
```

Обратите внимание, что *изнутри* процедуры все внешние (глобальные) метки видны, и при конфликте имён, к сожалению, предпочтение отдаётся *глобальной* метке, например:

```
MyProc proc
    jmp L; Будет переход на глобальную L:
L:
MyProc endp
. . .
L:: Будет переход сюда !!!
```

Создаётся впечатление, что программисты, создавшие компилятор MASM-6.14, имеют смутное представление о локализации имён в блоках, своё обучение программированию они явно начинали не с языка Паскаль 😊. Исходя из этого рекомендуется, например, локальные метки в процедурах, в отличие от меток в основной программе, начинать с символа @, например:

```
MyProc proc
    jmp @L
. . .
@L:
MyProc endp
```

В Ассемблере разрешена также так называемая *анонимная* метка @@:, такие метки могут встречаться в программе в любом количестве. Любая команда перехода **jmp** @F производит переход на ближайшую метку @@: вниз (**F**orward – вперёд), а команда **jmp** @B соответственно на ближайшую метку @@: вверх (**B**ack – назад).

По замыслу авторов Ассемблера, это должно резко сократить количество уникальных меток в программе.¹ Это некоторый аналог "безметочного" перехода относительно текущего значения счётчика размещения \$ (об этом говорилось ранее при изучении команд переходов), например

```
jmp $-8; EIP:=EIP-8-<длина команды jmp $-8>
```

Учтите также, что имя процедуры имеет тип метки, хотя за ним и не стоит двоеточие, как это принято для меток команд. Вызов процедуры обычно производится командой **call**, а возврат из процедуры – командой **ret**. Отметим, что команду **ret**, вообще говоря, можно использовать и *вне* тела процедуры, в этом случае компилятор Ассемблера выбирает команду *близкого* (**ret**n) возврата.

Необязательные спецификации процедуры определяют особенности её использования. Наиболее полезные спецификации будут изучены позже, после первого знакомства с программированием процедур на Ассемблере.

¹ Заметка на будущее: не стоит ставить анонимные метки внутри макроопределений, так как неизвестно, в какое место программы будет подставляться соответствующее макрорасширение.

По сравнению с Паскалем, само понятие процедуры в Ассемблере резко упрощается. Чтобы продемонстрировать это, рассмотрим такой синтаксически правильный фрагмент программы:

```
      mov  ax, 1
F      proc
      add  ax, ax
F      endp
      sub  ax, 1
```

Этот фрагмент полностью эквивалентен таким трём командам:

```
      mov  ax, 1
F:     add  ax, ax
      sub  ax, 1
```

Другими словами, описание процедуры на Ассемблере может встретиться в любом месте секции кода, это просто некоторый набор предложений, заключённый между директивами начала **proc** и конца **endp** описания процедуры. Заметим, что примерно такие же процедуры были в некоторых первых примитивных языках программирования высокого уровня, например, в начальных версиях языка Basic.

Аналогично, команда **call** является просто особым видом команды безусловного перехода, и может не иметь никакого отношения к описанию процедуры. Например, рассмотрим следующий синтаксически правильный фрагмент программы:

```
L:  mov  ax, 1

      . . .
      call L
```

Здесь вместо имени процедуры в команде **call** указана обычная метка. И, наконец, отметим, что в самом языке машины уже нет никаких процедур и функций, и программисту приходится моделировать все эти понятия.

Изучение программирования процедур на Ассемблере начнем со следующей простой задачи, в которой "напрашивается" использование процедур. Пусть надо ввести массивы X и Y знаковых целых чисел размером в двойное слово, массив X содержит 100 чисел, а массив Y содержит 200 чисел. Затем необходимо вычислить величину Sum:

$$\text{Sum} := \sum_{i=1}^{100} X[i] + \sum_{i=1}^{200} Y[i]$$

Переполнение результата за размер двойного слова при сложении элементов массивов будем для простоты игнорировать (т.е. выводить неправильный ответ без выдачи диагностики). Для данной программы естественно сначала реализовать процедуру суммирования элементов массива и затем дважды вызывать эту процедуру соответственно для массивов X и Y. Текст процедуры суммирования, как и в Паскале, будем располагать *перед* текстом основной программы (первая выполняемая команда программы, как известно, помечена меткой, указанной в директиве **end** модуля), хотя при необходимости процедуру можно было бы располагать и в конце программной секции, после макрокоманды **exit**.

Заметим, что, вообще говоря, будет описана *функция*, но в языке Ассемблера, как уже упоминалось, различия между процедурой и функцией не синтаксическое, а, скорее, семантическое. Другими словами то, что это функция, а не процедура, знает программист, но не компилятор Ассемблера, поэтому далее будет использоваться обобщенный термин *процедура*.

Перед тем, как писать процедуру, необходимо составить *соглашение о связях* (calling conventions) между основной программой и процедурой. Здесь необходимо уточнить, что под *основной программой* имеется в виду то место программы, где процедура вызывается по команде **call**. Таким образом, вполне возможен и случай, когда одна процедура вызывает другую, в том числе и саму себя, используя прямую или косвенную рекурсию.

Соглашение о связях между основной программой и процедурой включает в себя место расположения и способ доступа к параметрам, способ возврата результата работы (для функции) и некоторую другую информацию. Так, для нашего последнего примера "договоримся" с процедурой, что адрес (не индекс!) начала суммируемого массива *двойных слов* будет перед вызовом процедуры записан в регистр EBX, а количество элементов – в регистр ECX. Сумма элементов массива при возврате из

процедуры должна находится в регистре EAX. При этих соглашениях о связях у нас получится следующая программа (для простоты вместо команд для ввода массивов указан только комментарий).

```
include console.inc
.data
    X    dd    100 dup(?)
    Y    dd    200 dup(?)
    Sum  dd    ?
.code
Summa proc
; соглашение о связях: ebx - адрес первого элемента
; ecx=количество элементов, eax - ответ (сумма)
    sub  eax,eax; сумма:=0
L: add  eax,[ebx]; прибавление текущего элемента
    add  ebx,4;   адрес следующего элемента
    loop L
    ret
Summa endp
start:
; здесь команды для ввода массивов X и Y
    mov  ebx,offset X; адрес начала X
    mov  ecx,100;      число элементов X
    call Summa
    mov  Sum,eax;      сумма массива X
    mov  ebx,offset Y; адрес начала Y
    mov  ecx,200;      число элементов Y
    call Summa
    add  Sum,eax;      сумма массивов X и Y
    outintln Sum
    exit
end start
```

Надеюсь, что Вы легко разберётесь, как работает эта программа. А вот теперь, если попытаться "один к одному" переписать эту Ассемблерную программу на язык Free Pascal, то получится примерно следующее:

```
program A(input,output);
var
    X: array[1..100] of Longint;
    Y: array[1..200] of Longint;
    ebx: ↑Longint; Sum,ecx,eax: Longint;
procedure Summa;
    label L;
begin
    eax:=0;
L:  eax := eax + ebx↑;
    ebx:=ebx+4; {так в Паскале нельзя}
    dec(ecx);
    if ecx<>0 then goto L
end;
begin {Ввод массивов X и Y}
    ecx:=100; ebx:=↑X[1]; {так в Паскале нельзя}1
    Summa; Sum:=eax;
    ecx:=200; ebx:=↑Y[1]; {так в Паскале нельзя}
    Summa; Sum:=Sum+eax; Writeln(Sum)
end.
```

Как видно, в этой программе используется очень плохой стиль программирования, так как неявными параметрами процедуры являются глобальные переменные, т.е. полезный механизм передачи параметров Паскаля просто не используется. В то же время именно хорошо продуманный аппарат

¹ В языке Free Pascal для этой цели можно использовать оператор присваивания `ebx:=@X[1]`

формальных и фактических параметров делает процедуры и функции таким гибким, эффективным и надежным механизмом в языках программирования высокого уровня. Если с самого начала решать задачу суммирования массивов на Паскале, а не на Ассемблере, надо бы, конечно, написать, например, такую программу:

```
program A(input,output);
  type Mas= array[1..200] of Longint;
  var X,Y: Mas;
      Sum: Longint;
  function Summa(var A: Mas, N: Longint): Longint;
    var i,S: Longint;
  begin S:=0;
    for i:= 1 to N do S:=S+A[i];
    Summa:=S
  end;
begin {Ввод в массив X 100 и массив Y 200 чисел}
  Sum:=Summa(X,100)+Summa(Y,200); Writeln(Sum)
end.
```

Это уже грамотно составленная программа на Паскале с использованием функции. Теперь надо научиться, так же хорошо писать программы с процедурами и на Ассемблере, однако для этого понадобятся другие, более сложные, соглашения о связях между процедурой и основной программой. Если Вы хорошо изучили программирование на Паскале, то должны знать, что грамотно написанная процедура получает все свои входные данные как фактические параметры и не использует внутри себя глобальных имён переменных и констант. Такого же стиля работы с процедурами надо, по возможности, придерживаться и при программировании на Ассемблере. При работе с процедурами на языке Ассемблера будут использоваться так называемые **стандартные соглашения о связях**.

6.10.1. Стандартные соглашения о связях

Соглашение бывает особенно трудно расторгнуть – когда мы говорим «да», поторопивишись.

Вадим Синявский

Сначала поймем необходимость существования некоторых *стандартных* соглашений о связях между процедурой и вызывающей её основной программой. Действительно, иногда программист просто не сможет, скажем, "договориться" с процедурой, как она должна принимать свои параметры. В качестве первого примера можно привести так называемые *библиотеки стандартных процедур*. В этих библиотеках собраны процедуры, реализующие полезные алгоритмы для некоторой предметной области (например, для работы с матрицами), так что программист может внутри своей программы вызывать нужные ему процедуры из такой библиотеки. Библиотеки стандартных процедур обычно поставляются в виде набора так называемых *объектных модулей*, что практически исключает возможность вносить какие-либо изменения в исходный текст этих процедур (с объектными модулями Вы познакомитесь в этой книге далее). Таким образом, получается, что взаимодействовать с процедурами из такой библиотеки можно, только используя те соглашения о связях, которые были использованы при создании этой библиотеки.

Другим примером является написание частей программы на различных языках программирования, при этом чаще всего основная программа пишется на некотором языке высокого уровня (Фортране, Паскале, С и т.д.), а процедура – на Ассемблере. Вспомним, что когда говорилось об областях применения Ассемблера, то одной из таких областей и было написание процедур, которые вызываются из программ на языках высокого уровня. Например, для языка Free Pascal такая, как говорят, *внешняя*, процедура может быть описана в программе на Паскале следующим образом:

```
procedure P(var x: Longint; y: Longint); external;
```

Служебное слово **external** является указанием на то, что эта функция описана не в данном программном модуле и Паскаль-машина должна вызвать эту *внешнюю* процедуру, написанную, вообще говоря, на другом языке, и как-то передать ей параметры.¹ Если программист пишет эту проце-

¹ На языке Free Pascal программист, как правило, должен также задать специальную директиву {\$L <имя файла>} в которой указывается расположение в файловой системе объектного модуля, содержащего внешние процедуры и функции.

дуру на Ассемблере, то он конечно никак не может "договориться" с компилятором Паскаля, как он хочет получать параметры, а для функции ещё и возвращать результат её работы.

Именно для таких случаев и разработаны **стандартные соглашения о связях**. При этом если процедура или функция, написанная на Ассемблере, соблюдает эти стандартные соглашения о связях, то это гарантирует, что эту процедуру или функцию можно будет вызывать из программы, написанной на другом языке программирования. Разумеется, в этом другом языке (более точно – в *системе программирования*, включающей в себя этот язык) тоже должны соблюдаться такие же стандартные соглашения о связях.

Рассмотрим типичные стандартные соглашения о связях между основной программой и процедурой, эти соглашения поддерживаются многими современными системами программирования. Обычно стандартные соглашения о связях включают в себя следующие пункты.

1. Соглашения об именах

Необходимо договориться о том, *каким* вызывающая программа "видит" имя процедуры. Внутри одного программного модуля этой проблемы не существует, так как и вызывающая программа и процедура находятся в одной области видимости. Проблема возникает, если вызывающая программа и процедура находятся в разных программных модулях и непосредственно не видят друг друга. Разные языки (точнее, разные системы программирования) имеют разные правила о том, как имя процедуры должно выглядеть "во внешнем мире", т.е. из других модулей. Например, имя функции `Summa` на Ассемблере может для других модулей иметь внешнее представление `_SUMMA@0`. Наиболее часто используемое соглашение о связях с именем **stdcall** предполагает неразличимость больших и малых букв имён во "внешнем мире". Подробно об этом будет говориться в главе, посвященной модульному программированию.

2. Соглашения о передаче параметров и возврате значения функции

Далее, необходимо договориться о том, как фактические параметры процедуры передаются на место формальных параметров. Как известно, в языке Паскаль параметры можно передавать в процедуру по значению и по ссылке, это верно и для многих других языков программирования. Заметим, однако, что в некоторых языках программирования одного из этих способов может и не быть. Так, в языке С (не С++) параметры передаются только по значению, а в первых версиях языка Фортран – только по ссылке. Ссылка фактически является адресом, в плоской модели памяти адрес имеет длину 32 бита и помещается на любой регистр общего назначения (EAX, EBX и т.д.) или в переменную формата двойного слова (**dd**). Параметры-значения в принципе могут иметь любую длину, однако в языках программирования настоятельно рекомендуется *большие* по размеру параметры передавать по ссылке. Небольшие (не более 32 бит) параметры-значения передаются в формате **dword**, расширяя, при необходимости, их размер до 32 бит. Будет ли это расширение знаковым или беззнаковым, естественно, зависит от типа этого параметра. Для передачи больших *параметров-значений* в системах программирования предусматриваются специальные правила.

Наиболее быстрой является передача параметров через регистры общего назначения, как это было сделано выше в функции `Summa`. Например, когда число параметров не превышает трёх, можно попросить Free Pascal передать параметры через регистры:

```
procedure P(var X: real; Y, Z: Longint);  
  register; external;1
```

В этом случае адрес `X` передаётся в процедуру на регистре EAX, а значения `Y` и `Z` соответственно на регистрах EDX и ECX. Когда параметров больше трёх, то остальные передаются через стек, о чём говорится далее. Для передачи вещественных параметров-значений дополнительно используются вещественные регистры.

Регистров, однако, очень мало, они к тому же интенсивно используются при работе программы, так что в общем случае этот способ малопригоден. Более универсальным является

¹ В 64-битных моделях число регистров для передачи адресных и целочисленных параметров увеличено до четырех. Для языка С вместо ключевого слова **register** при передаче параметров через регистры используется похожий способ передачи **fastcall** (правда, используются только два регистра).

другой способ передачи параметров, когда фактические параметры перед вызовом процедуры или функции записываются в стек.¹

При передаче параметра *по значению* в стек записывается само это значение, при необходимости преобразованное в формат двойного слова, а в случае передачи параметра *по ссылке* в стек записывается адрес начала фактического параметра.

Порядок записи фактических параметров в стек может быть *прямым* (сначала записывается первый параметр, потом второй и т.д.) или *обратным* (когда, наоборот, сначала записывается последний параметр, потом предпоследний и т.д.). В разных языках программирования этот порядок различный. Так, в языке С по умолчанию это *обратный* порядок, а в языке Фортран – *прямой*.² Во многих языках программирования при описании процедуры или функции можно явно указать (изменить) способ передачи параметров. Например, в языке Free Pascal способ передачи параметров можно указать в заголовке процедуры так:

```
procedure P(var X: real; Y, Z: Longint);  
pascal; external;
```

задаёт прямой способ передачи параметров, а

```
procedure P(var X: real; Y, Z: Longint);  
stdcall; external;
```

задаёт обратный способ передачи параметров во внешнюю процедуру.

Функция возвращает своё значение в регистрах AL, AX, EAX или в паре регистров <EDX, EAX>, в зависимости от величины (типа) этого значения. Вещественный результат возвращается на вещественном регистре st0 (R0). Для возврата значений, превышающих учетверённое слово, устанавливаются специальные соглашения.

Перед возвратом из процедуры и функции стек очищается от всех фактических параметров. Напомним, что в языке Паскаль *формальные* параметры, в которые передаются соответствующие им *фактические* параметры, являются *локальными* переменными процедур и функций, и, как любые локальные переменные, должны быть уничтожены при выходе из процедуры и функции. Исключением является особый способ передачи параметров, принятый в языке С, который задаётся спецификацией **cdecl** (сокращение от C-declaration), например:

```
function F(var X: real; Y: char):char;  
cdecl; external;
```

Передача параметров производится так же, как и в случае спецификации **stdcall**, однако *очистку* стека от фактических параметров производит не вызванная процедура (перед возвратом), а вызывающая программа (после возврата). Этот способ полезен только при передаче в процедуру и функцию *переменного* числа фактических параметров, в основном он применяется в языке С. В вызывающей программе, в отличие от процедуры, легче определить, сколько параметров записано в стек для конкретного вызова. Плохо в этом случае то, что такое удаление приходится выполнять не в одном месте (в конце описания процедуры), а в *каждой* точке возврата из процедуры, что увеличивает размер кода (в программе может быть очень много вызовов одной и той же процедуры)

3. Соглашения о локальных переменных

Если в процедуре или функции необходимы *локальные* переменные (в смысле языков высокого уровня), то память для них отводится в стеке. Обычно это делается путем записи в стек некоторых величин, или же сдвигом указателя вершины стека, для чего надо *уменьшить* значение регистра ESP на число байт, которые занимают эти локальные переменные. Как и для фактических параметров, при необходимости размер памяти под локальную переменную округляется до двойного слова. Таким образом, символьная переменная занимает в стеке 4 байта, 3 из которых не используются. Другими словами, локальный массив из четы-

¹ Большинство современных процессоров имеют аппаратно реализованный стек, если же это не так, то стандартные соглашения о связях для таких ЭВМ устанавливаются каким-нибудь другим способом, здесь этот случай рассматриваться не будет.

² Обратный порядок передачи параметров применяется и в Windows при использовании так называемых системных вызовов (видимо, это отзвук того, что большая часть Windows написана на языке С). Впрочем, чтобы не путать программистов на Ассемблере, обычно при обращении к системному вызову используются макроккоманды, параметры в которых идут в обычном порядке.

рѐх символов `array[1..4] of char`; можно разместить в стеке "за ту же цену", что и одну символьную переменную.

Перед возвратом из процедуры и функции стек очищается от всех локальных переменных, напомним, что в языках высокого уровня это стандартное действие при выходе из процедуры и функции (в общем случае из блока).

4. Соглашения об использовании регистров

Если в процедуре или функции изменяются какие-либо регистры, то в начале работы необходимо запомнить значения этих регистров в локальных переменных (т.е. в стеке), а перед возвратом – восстановить эти значения. В плоской модели памяти нельзя (даже временно) изменять сегментные регистры. Для функции, естественно, не запоминаются и не восстанавливаются регистр(ы), на котором(ых) возвращается результат её работы. Обычно также принято не запоминать и не восстанавливать регистр флагов и регистры для работы с вещественными числами.¹

Участок стека, в котором при *каждом* вызове процедуры или функции размещаются их фактические параметры, локальные переменные и адрес возврата, называется *стековым кадром* (stack frame). Стекковый кадр начинает строить основная программа перед каждым вызовом процедуры, помещая туда (т.е. записывая в стек) фактические параметры. Затем команда **call** помещает в стек адрес возврата (это двойное слово) и передаёт управление на начало процедуры. Далее уже сама процедура продолжает построение стекового кадра, размещая в нём свои локальные переменные, в частности, для хранения старых значений изменяемых регистров.

Заметим, что если построением стекового кадра занимаются как основная программа, так и процедура, то полностью разрушить (очистить) стековый кадр должна сама процедура, так что при возврате в основную программу стековый кадр будет полностью уничтожен.²

При программировании на Ассемблере MASM передача параметров будет проводиться по спецификации **stdcall**. В языке Free Pascal этот режим задаётся директивой **{\$Calling stdcall}** или же заданием при объявлении внешней процедуры слова-модификатора **stdcall**:

```
procedure P(var X: real; Y: char);
stdcall; external;
```

В Ассемблере это тоже наиболее часто используемый режим передачи параметров, это объясняется тем, что системные функции операционных систем (Windows, Unix, да и Android), к которым обращаются макрокоманды, написаны на языке C.

Перепишем теперь нашу последнюю "хорошую" программу суммирования массивов с языка Паскаль на Ассемблер с использованием стандартного соглашения о связях. Описание функции поместим в начало секции кода, до метки `start`. Ниже показано возможное решение этой задачи, подробные комментарии сразу после текста программы.

```
include console.inc
.data
    X    dd 100 dup(?)
    Y    dd 200 dup(?)
    Sum  dd ?
.code
Summa proc
; стандартные соглашения о связях
    push ebp
    mov  ebp, esp; база стекового кадра
    sub  esp, 4;   порождение локальной переменной S
    push ebx;      запоминание используемых
    push ecx;      регистров ebx и ecx
S    equ  dword ptr [ebp-4]
; S будет именем локальной переменной в стеке
    mov  ecx, [ebp+12]; ecx:=длина массива N
```

¹ Системные вызовы функций API Microsoft Windows должны запоминать и восстанавливать регистры ESI, EDI, EBP и EBX, на EAX возвращается результат функции, сохранность остальных регистров (ECX и EDX) не гарантируется. В регистре флагов не рекомендуется возвращать DF=1, об это флаге будет говориться далее.

² Как уже говорилось, однако, при спецификации **cdecl** (как в языке C) удаление из стека фактических параметров делает не процедура, а вызывающая программа.

```

    mov     ebx, [ebp+8];    ebx:=адрес первого элемента
    mov     S, 0;           сумма:=0
L:  mov     eax, [ebx];     нельзя add S, [ebx] - нет формата
    add     S, eax
    add     ebx, 4;         i:=i+1
    loop    L
    mov     eax, S;         результат функции на eax
    pop     ecx
    pop     ebx;           восст. регистров ecx, ebx
    mov     esp, ebp;       уничтожение локальной переменной S
    pop     ebp;           восст. регистра ebp
    ret     2*4;           возврат с очисткой стека от двух параметров
Summa endp
; основная программа
start:
; здесь команды для ввода массивов X и Y
    push    100;           второй фактический параметр
    push    offset X;      первый фактический параметр
    call    Summa
    mov     Sum, eax;       сумма массива X
    push    200;           второй фактический параметр
    push    offset Y;      первый фактический параметр
    call    Summa
    add     Sum, eax;       сумма массивов X и Y
    outintln Sum
    exit
end start

```

Подробно прокомментируем эту программу. Первый параметр функции передаётся по ссылке, а второй – по значению, эти параметры записываются в стек в обратном порядке. После выполнения команды вызова процедуры **call Summa** стековый кадр имеет вид, показанный на рис. 6.4. После полного формирования стековый кадр будет иметь вид, показанный на рис. 6.5. (справа на этом рисунке показаны смещения слов в стеке относительно значения регистра ЕВР).

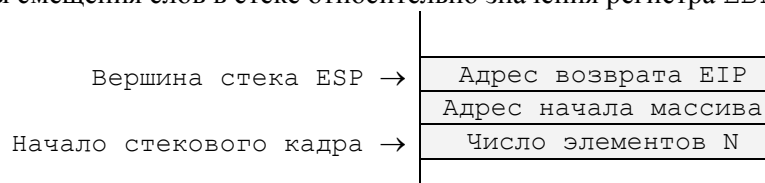


Рис. 6.4. Вид стекового кадра при входе в функцию Summa.

Отметим далее то значение, которое отводится индексному регистру ЕВР при работе со стеком. Этот регистр используется в процедуре для хранения *базы* стекового кадра для доступа к параметрам и локальным переменным процедуры или функции. Так и было сделано в этой программе, когда регистр ЕВР (по сути, это ссылочная переменная в смысле Паскаля) был поставлен примерно на середину стекового кадра. Теперь, отсчитывая смещения от значения регистра ЕВР вниз, например [ЕВР+8], получаем доступ к фактическим параметрам, отсчитывая смещение вверх – доступ к сохраненным значениям регистров и локальным переменным. Например, выражение [ЕВР-4] определяет адрес локальной переменной, которая в программе на Ассемблере, повторяя программу на Паскале, названа именем S (см. рис. 6.4). Теперь понятно, почему регистр ЕВР называют *базой* (base pointer) стекового кадра.

Итак, при входе в процедуру сначала в стеке запоминается регистр ЕВР, затем порождаются необходимые локальные переменные, а потом спасаются используемые регистры. Эти действия носят название *пролога* (prolog) процедуры.

Обратите внимание, что локальные переменные в стековом кадре не имеют имён, что может быть не совсем удобно для программиста. В нашем примере локальной переменной присвоено имя S при помощи директивы эквивалентности **equ** (S – текстовая макропеременная)

```
S equ dword ptr [ebp-4]
```

И теперь всюду вместо имени *S* компилятор Ассемблера будет подставлять текстовую строку (адресное выражение) **dword ptr** [ЕВР-4], которое имеет, как и нужно, тип двойного слова, расположенного в стеке. Для порождения этой локальной переменной ей отводится место в стеке с помощью команды

sub esp, 4; порождение локальной переменной

т.е. просто уменьшается значение регистра-указателя вершины стека на 4 байта. Эта переменная порождается, как и в Паскале, с *неопределённым* начальным значением. Этой же цели можно было бы достичь, например, более короткой (но менее понятной для читающего программу человека) командой

push eax; порождение локальной переменной ?

или же такой экзотической командой

add esp, -4; порождение локальной переменной !

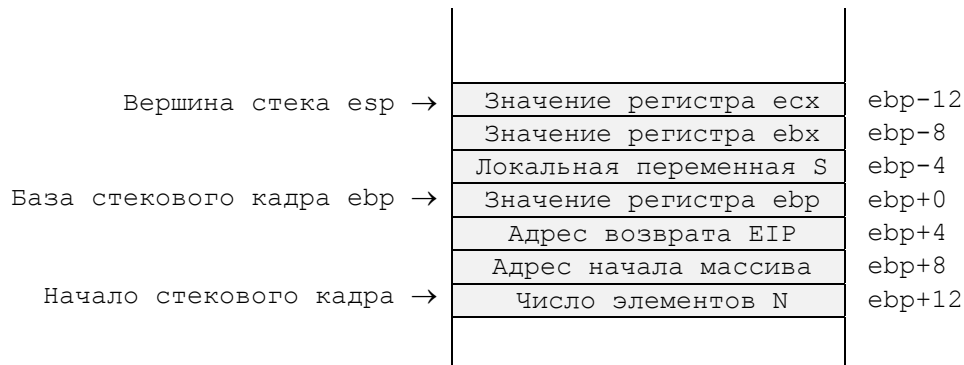


Рис. 6.5. Вид полного стекового кадра (справа показаны смещения слов кадра относительно значения регистра ЕВР).

Перед возвратом из функции началось разрушение стекового кадра, как этого требуют стандартные соглашения о связях. Сначала из стека восстанавливаются старые значения регистров ЕСХ и ЕВХ, затем командой

mov esp, ebp; уничтожение локальной переменной *S*

уничтожается локальная переменная *S* (т.е. она удаляется из стека, т.к. в регистре ЕВР записано старое значение регистра ЕСП, каким оно было до порождения локальных переменных). Далее восстанавливается регистр ЕВР (заметьте, база стекового кадра не понадобится далее в нашей функции). И, наконец, команда возврата

ret 2*4; возврат с очисткой стека

удаляет из стека адрес возврата и два двойных слова – значения фактических параметров функции. Теперь уничтожение стекового кадра завершено. Все эти действия носят название *эпилога* (epilogue) процедуры.

Важной особенностью использования стандартных соглашений о связях является и то, что они позволяют производить *рекурсивный* вызов процедур и функций, причём рекурсивный и не рекурсивный вызовы "по внешнему виду" не отличаются друг от друга. В качестве примера рассмотрим реализацию функции вычисления факториала от неотрицательного (т.е. беззнакового) целого числа, при этом будем предполагать, что значение факториала поместится в двойное слово (иначе будет выдаваться неправильный результат без диагностики об ошибке). На языке Паскаль эта функция имеет следующий вид:

```
function Fact (N: Longword) : Longword;
begin
  if N<=1 then Fact:=1
    else Fact:=N*Fact (N-1)
end;
```

Будем надеяться, что язык Паскаль Вы все хорошо знаете, и без пояснений понимаете, как работает рекурсивная функция 😊. Реализуем теперь этот алгоритм в виде функции на Ассемблере. Сначала заметим, что условный оператор Паскаля

```
if N<=1 then Fact:=1
else Fact:=N*Fact (N-1)
```

для программирования на Ассемблере неудобен, так как содержит две ветви (**then** и **else**), которые придётся размещать в линейной структуре машинной программы, что повлечёт использование между этими ветвями команды *безусловного* перехода. Поэтому лучше преобразовать этот оператор в такой эквивалентный вид:

```
Fact:=1;
if N>1 then Fact:=N*Fact(N-1)
```

Теперь приступим к написанию функции Fact на Ассемблере:

```
Fact proc; стандартные соглашения о связях
    push ebp
    mov ebp,esp; база стекового кадра
; Локальных переменных нет
    push edx; спасаем регистр edx
N equ dword ptr [ebp+8]; фактический параметр N
    mov eax,1; Fact при N<=1
    cmp N,1
    jbe Vozv
    mov eax,N
    dec eax
    push eax; Параметр N-1
    call Fact; Рекурсия
    mul N; <edx,eax>:=eax*N = Fact(N-1)*N
Vozv:
    pop edx; восст. регистр edx
    pop ebp; восст. регистр ebp
    ret 4
Fact endp
```

В качестве примера рассмотрим вызов этой функции для вычисления факториала `Writeln(Fact(5))`. Такой вызов можно в основной программе сделать, например, следующими командами:

```
push 5
call Fact
outwordln eax
```

На рис. 6.6 показан вид стека после того, как произведён первый *рекурсивный* вызов функции, заметьте, что в стеке при этом два стековых кадра, первый соответствует вызову функции из основной программы.

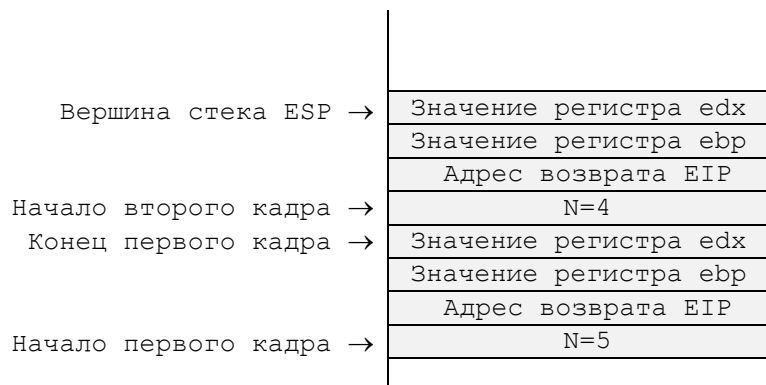


Рис. 6.6. Два стековых кадра функции Fact.

В качестве ещё одного примера рассмотрим реализацию с помощью *процедуры* следующей задачи. Задана константа `N=300000`, найти количество элементов массива содержащего `N` знаковых целых чисел, размером в слово, которые больше последнего элемента этого массива. На языке Free Pascal эту процедуру можно записать, например, следующим образом:

```
const N=300000;
type Mas=array[1..N] of Longint;
var X: Mas; K: Longword;
procedure NumGtLast(var X: Mas; N: Longword; var K: Longword);
var i: Longword;
begin
```



```

    K:=0;
    for i:=1 to N do
        if X[i]>X[N] then inc(K)
    end;
    begin P(X,N,K) end.

```

Перед реализацией процедуры NumGtLast напечатаем сначала фрагмент на Ассемблере для вызова этой процедуры:

```

push offset K; Адрес K
push N;        Длина массива
push offset X; Адрес массива X
call NumGtLast

```

При входе в процедуру стековый кадр будет иметь вид, показанный на рис. 6.7.

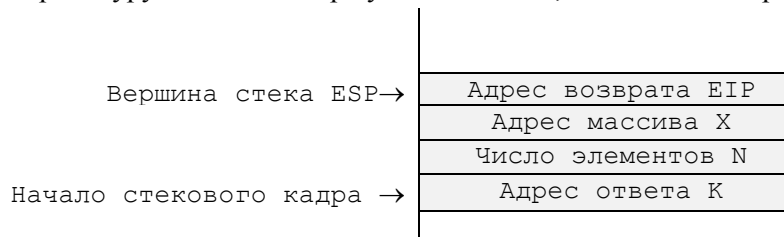


Рис. 6.7. Стековый кадр при входе в процедуру NumGtLast.

Теперь опишем саму процедуру:

```

NumGtLast proc
    push ebp
    mov  ebp,esp; база стекового кадра
; сохранение регистров
    push eax
    push ebx
    push ecx
    push edi
; локальная переменная K на регистре ebx
    sub  ebx,ebx; K:=0
    mov  ecx,[ebp+12];    длина массивов N
    mov  edi,[ebp+8];     адрес массива X
    mov  eax,[edi+4*ecx-4]; это eax:=X[N]
    dec  ecx; N:=N-1 последний эл-нт не смотрим
L:  cmp  [edi],eax;      X[i]>X[N] ?
    jle  L1; if X[i]<=X[N] then goto L1
    inc  ebx;    K:=K+1
L1: add  edi,4;        i:=i+1
    loop L
    mov  edi,[ebp+16]; адрес K
    mov  [edi],ebx;    послать ответ в K
; восстановление регистров
    pop  edi
    pop  ecx
    pop  ebx
    pop  eax
    pop  ebp
    ret  3*4;    Очистка от 3-х параметров
NumGtLast endp

```

Обратите внимание на типичную ошибку использования параметра-результата, переданного по ссылке. Вместо приведённых команд

```

mov  edi,[ebp+16]; адрес K
mov  [edi],ebx;    послать ответ в K

```

используют неправильную запись ответа в переменную K в виде

```

mov  [ebp+16],ebx; Это (адрес K) :=ответ

```

Другими словами, ответ пишется не в область данных, а в область стека, где этот ответ, кстати, будет уничтожен при возврате из процедуры.

Итак, внутри каждой процедуры надо написать пролог и эпилог, а перед вызовом этой процедуры записать в стек фактические параметры. Так как это достаточно типовые фрагменты программ на Ассемблере, то возникает желание автоматизировать этот процесс, поручив делать это самому компилятору с Ассемблера.¹ Именно так и происходит при программировании на языках высокого уровня, где, скажем, компилятор Паскаля сам формирует на машинном языке вызов, пролог и эпилог каждой процедуры.

Однако, чтобы такую же работу делал компилятор Ассемблера, ему надо иметь информацию о формальных параметрах, локальных переменных и используемых регистрах. Всю эту дополнительную информацию на языке Ассемблера MASM можно включить в описание процедуры.

Рассмотрим, как это делается. Возьмём в качестве примера простую программу с процедурой на языке Free Pascal:

```
var X: Longint; Y: Word;
procedure P(var Par1: Longint; Par2: Word);
  var Loc1, Loc2: Longword;
  begin
    Тело процедуры
  end;
begin P(X,Y) end.
```

Пусть в своей работе процедура использует регистры EAX и EBX, которые необходимо сохранить при входе и восстановить при выходе. Описание процедуры на Ассемблере может выглядеть таким образом:

```
P  proc
  push ebp
  mov  ebp,esp; база стекового кадра
; порождение переменных Loc1 и Loc2
  sub  esp,8
; сохранение регистров
  push eax
  push ebx
comment *
  Можно задать такие директивы эквивалентности
  для удобной работы с параметрами и локальными
  переменными по именам, а не по адресам:
Loc1 equ dword ptr [ebp-4]
Loc2 equ dword ptr [ebp-8]
Par1 equ dword ptr [ebp+8]
Par2 equ word ptr  [ebp+12]
*
  Тело процедуры
;
; восстановление регистров
  pop  ebx
  pop  eax
; уничтожение Loc1 и Loc2
  mov  esp,ebp; это короче, чем sub esp,8
  ret  8; очистка 2-х параметров
P  endp
```

Перепишем теперь описание процедуры P, задав для компилятора Ассемблера дополнительную информацию:

```
P  proc uses eax ebx, Par1:dword, Par2:word
  local Loc1,Loc2:dword
comment * Начало большого комментария
; Ассемблер автоматически вставляет сюда команды:
  push ebp
  mov  ebp,esp; база стекового кадра
```

¹ В язык машины введены специальные команды: **enter** и **leave** для создания пролога и эпилога, но они предоставляют мало возможностей и неудобны для программиста, пользоваться ими не рекомендуется.

```

; порождение переменных Loc1 и Loc2
sub esp,8; порождение Loc1 и Loc2
; сохранение регистров
push eax
push ebx
; Ассемблер автоматически вставляет директивы
Loc1 equ dword ptr [ebp-4]
Loc2 equ dword ptr [ebp-8]
Par1 equ dword ptr [ebp+8]
Par2 equ word ptr [ebp+12]
; это локальные имена, невидимые извне процедуры !

```

Тело процедуры

```

; вместо ret автоматически вставляются команды:
; восстановление регистров
pop ebx
pop eax
; уничтожение Loc1 и Loc2
mov esp,ebp; это короче, чем sub esp,8
ret 8; очистка 2-х параметров
* Конец большого комментария
; ret; останется в тексте как комментарий
P endp

```

Таким образом, описание процедуры приобретает компактный вид:

```

P proc uses eax ebx, Par1:dword, Par2:word
local Loc1,Loc2:dword
;
;
;
;
ret
P endp

```

Тело процедуры

Вызов этой процедуры P (X, Y) на Ассемблере выглядит так:

```

movzx eax,Y;    eax:=Longword(Y)
push  eax;      Параметр Par2
push offset X;  Параметр Par1
call  P

```

В принципе, в языке Ассемблера есть и удобное средство для автоматизации вызова процедур. Для этого можно использовать стандартную макрокоманду,¹ имеющую служебное имя **invoke**:

```
invoke P,offset X,Y
```

Заметьте, что в макрокоманде **invoke** параметры следуют в привычном для программиста *прямом* порядке, кроме того, эта макрокоманда настраивается на режимы **stdcall** или **cdecl** и в режиме **cdecl** сама очищает стек после возврата из процедуры. Такой вызов процедуры нельзя делать, если процедура описана *после* макрокоманды **invoke** или, что бывает много чаще, описана в *другом модуле*, и, таким образом, не видна компилятору Ассемблера. В языках высокого уровня в этом случае перед вызовом процедуры необходимо задать *объявление* этой процедуры. Вы должны знать, как это делается, например, на Паскале. Аналогичное объявление процедуры на Ассемблере делается в виде так называемого *прототипа* процедуры:

```
P proto :dword, : word
```

¹ По аналогии со стандартной макрокомандой **invoke**, вставка в процедуру команд для пролога и эпилога реализована посредством вызова компилятором Ассемблера *стандартных* макрокоманд с именами **PrologueDef** и **EpilogueDef**. Можно запретить вызывать эти макрокоманды с помощью опций Ассемблера **option Prologue:NONE** и **option Epilogue:NONE**. Программист может также *переопределить* **PrologueDef** и **EpilogueDef**, написав свои *собственные* макроопределения с такими же именами (список параметров этих макроопределений приведён в документации по Ассемблеру MASM).

В прототипе указано имя процедуры и размер всех её параметров. Если процедура описана в данном модуле, её имя считается входной точкой (**public**), а если не описана, то внешним именем (**extrn**). Используя описание процедуры или её прототип, служебная макрокоманда **invoke** подставит вместо себя приведённые ранее четыре команды для вызова процедуры P.¹

Студенты, изучающие данный курс, должны уметь сами писать все необходимые команды для реализации вызова, пролога и эпилога процедур и функций.²

Усложнение описания процедуры и использования служебной макрокоманды для вызова процедур являются типичными примерами *повышения уровня языка* программирования. Такое повышение может делаться как усложнением самого языка, так и активным использованием макрокоманд. Вспомним, что, например, макрокоманда **outint X,10** отличается от оператора стандартной процедуры **Write(X:10)** Паскаля чисто синтаксически.

Из описания реализации процедур видно, что аппаратный стек используется очень интенсивно, особенно область, близкая к вершине стека. Для повышения эффективности, начиная с процессоров Pentium 4 NetBurst, реализован особый механизм RAS (Return Address Stack), при этом область, близкая к вершине стека, кэшируется в быстродействующей памяти (фактически на регистрах).

6.11. Дальние процедуры на Ассемблере

Шарлатан тот, кто не может простыми словами объяснить, чем он занимается.

Курт Воннегут

Наряду с близким, в языке машины есть также *дальний* (межсегментный) вызов процедуры **call op1**, с параметром форматов m48=m16:m32 (**df, fword**) или i48=i16:i32, этот операнд обычно обозначается как **seg:offset**. Для понимания работы дальних вызовов предварительно нужно изучить так называемые *уровни привилегий*. Это очень сложная тема, однако можно отметить, что в рамках данного курса команды дальнего вызова процедур не используются в программировании на Ассемблере, с ними работают только служебные программы для вызова системных процедур Windows. Так что это теоретическая тема, необходимая, однако, для понимания архитектуры изучаемого компьютера.

6.11.1. Уровни привилегий

В этом мире люди ценят не права, а привилегии.

Генри Луис Менкен



Уровни привилегий призваны обеспечить безопасность функционирования вычислительной системы, они определяют возможность выполнения тех или иных команд, в частности, доступ к сегментам кода и данных. Система привилегий защищает данные и команды от несанкционированного доступа, т.е. от случайного или преднамеренного неправомерного чтения, записи или выполнения. В этой книге рассматривается только защита на уровне привилегированных команд и сегментов, третий вид защиты (на уровне страниц виртуальной памяти) полностью изучается в курсе по операционным системам.

Уровень привилегий определяется текущим режимом работы процессора в *защищённом режиме*, соответственно различают привилегированный и непривилегированный режимы работы. Режим работы процессора для текущей программы CPL (Current Privilege Level) хранится в разрядах 0 и 1 селектора в кодовом сегментном регистре CS, и может принимать значения 0, 1, 2 и 3, эти уровни

¹ Только в стандартной макрокоманде **invoke** вместо параметра **offset** X адрес переменной можно задавать в виде **ADDR X**. У этого второго способа есть свои достоинства и недостатки. У формы параметра **ADDR X** имя X не может содержать ссылку вперёд, т.е. быть описанным где-то далее по тексту программы, зато X может быть именем локальной переменной в стеке, что для формы параметра **offset** X невозможно.

² В следующих 64-битных моделях ЭВМ программисту на Ассемблере при написании и вызовах процедур *настоятельно рекомендуется* использовать только встроенные средства Ассемблера. В то же время *понимать*, как это выглядит на языке машины, является *совершенно необходимым* для уровня университетского образования.

называются кольцами (Ring) защиты. Операционная система работает в кольце Ring-0 с CPL=0, а программы пользователя в кольце Ring-3 с CPL=3, остальные два уровня Windows (и почти все другие ОС) не использует.

Как будет подробно рассказано в главе, посвященной мультипрограммному режиму работы ЭВМ, все команды машины делятся на обычные (команды пользователя) и привилегированные (запрещённые для обычного пользователя). Для выполнения привилегированных команд программа должна работать в привилегированном режиме с CPL=0.¹ Обратите внимание, когда говорится, что уровень привилегий одной программы *больше* (или выше), чем у другой, то CPL первой программы *меньше*, чем у второй, что непривычно для человека.

Режим работы используется не только для контроля выполнения привилегированных команд, но и как механизм защиты селекторов и сегментов. Например, в *селекторах* сегментов есть поле RPL (Request Privilege Level – требуемый уровень привилегий задачи для использования данного селектора). В дескрипторах тоже имеется аналогичное поле DPL (Descriptor Privilege Level – требуемый уровень привилегий задачи для доступа к этому дескриптору, точнее, к объекту, описываемому этим дескриптором). Селектор может указывать на различные объекты (сегменты, шлюза вызова, задачи, локальную дескрипторную таблицу и т.д.). RPL селектора хранится в разрядах 0 и 1 селектора; DPL дескриптора, хранится в поле DPL дескриптора. Отметим, что CPL является частным случаем RPL, т.к. он хранится в селекторе регистра CS. Таким образом, у всех важных объектов есть свой уровень привилегий.

Для доступа к *сегменту данных* должно выполняться условие $\max(CPL, RPL) \leq DPL$, значение $\max(CPL, RPL)$ называется *эффективным уровнем привилегий* (соображаем, что CPL в регистре CS, RPL в регистрах DS или ES, DPL в дескрипторе сегмента данных). Для доступа к сегменту стека требуется более жёсткое условие $CPL=RPL=DPL$, т.е. CPL в регистре CS равен RPL в регистре SS и равен DPL в дескрипторе сегмента стека. Таким образом, для работы привилегированной программы ей обязательно требуется системный стек с тем же уровнем привилегий. Правила проверки привилегий доступа к сегментам кода будут определены далее при описании дальнего вызова процедур.

Например, пусть $CPL=RPL=3$, а у дескриптора кодового сегмента $DPL=0$, тогда программа пользователя выполняет процедуру из кодового сегмента операционной системы (иногда бывает и так). Для случая $CPL=0$, а у селектора сегментного регистра данных DS $RPL=3$ и у соответствующего дескриптора сегмента данных $DPL=3$, тогда операционная система работает с сегментом данных пользователя. На рис. 6.8 показаны правила доступа из кодового сегмента к сегментам кода, данных и стека с разным уровнем привилегий.^{iv}

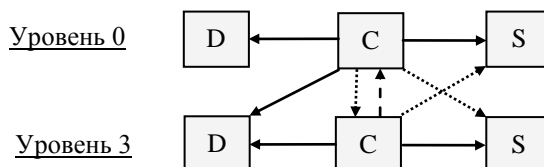


Рис 6.8. Доступ из кодового сегмента к другим сегментам с разным уровнем привилегий (сегменты: D – данных, C – кода, S – стека). Линии: сплошная – разрешено, точками – не разрешено, пунктир – только к согласованному (подчинённому) сегменту.

Как видно, это *двухуровневая* система управления доступом, право доступа к селектору объекта не обеспечивает автоматического права доступа к самому объекту (право войти в некоторый лифт само по себе не даёт права выйти на любом этаже, куда едет этот лифт, ведь этим лифтом пользуются и простые граждане и начальники 😊).

Вернёмся теперь к изучению дальнего вызова процедур. Дальний вызов процедуры имеет вид

call op1 = **call** seg:offset

и выполняется по следующей схеме:

¹ Заметим, однако, что привилегия на выполнение команд ввода/вывода (**sti**, **cli**, **in**, **out** и некоторых других), определяется не CPL, а полем IOPL (I/O Privilege Level – уровень привилегий ввода-вывода) в 12-ом и 13-ом битах регистра флагов EFLAGS, а также так называемой картой ввода/вывода. Таким образом, команды ввода/вывода могут быть разрешены для выполнения программой пользователя, в то время как остальные привилегированные команды запрещены.

Встек (EIP) ; Встек (Longword (CS)) ; <Переход на процедуру>

Поле *seg* содержит *селектор* объекта перехода, который может задавать:

- дескриптор кодового сегмента процедуры, тогда это *прямой вызов*,
- дескриптора шлюза вызова (Call Gate) процедуры, тогда это *вызов через шлюз*,
- дескриптор задачи TSS (Task State Segment), тогда это *переключение задач*.

Сначала надо понять, что это передача управления в *другой* сегмент кода, а так как в плоской модели памяти у программы пользователя только *один* сегмент кода, то это вызов *чужой* процедуры, чаще всего системный вызов (system call) функции операционной системы. Для задания дальнего вызова в Ассемблере записывается прямое указание `call far ptr op1`.

При прямом вызове селектор *seg* задаёт дескриптор кодового сегмента, в котором располагается вызываемая процедура. Большое значение при этом имеет бит с именем C (Conforming) в этом дескрипторе, это так называемый бит подчинённости (согласованности). Когда этот бит C=0 (эти кодовые сегменты называются неподчинёнными или *несогласованными*), то переход в такой сегмент командой **call** возможен только из сегмента, уровень привилегий которого не меньше привилегий целевого сегмента, т.е. CPL≤DPL. Можно сказать, что процедуры там "только для своих", и "менее знатные" гости туда просто не допускаются. Так блокируется несанкционированный вызов процедур операционной системы из программ обычного пользователя. Для случая C=1 кодовый сегмент называется подчинённым или *согласованным*, процедуру из такого сегмента всем разрешается вызывать, однако выполняться она будет с уровнем привилегий вызывающего сегмента, т.е. CPL≥DPL и CPL не меняется (в гости пустили, но делать всё, что можно хозяевам, нельзя). Переход на начало вызванной процедуры **jmp op1=seg:offset** при прямом вызове производится по правилу

CS:=seg; EIP:=offset

Как видно в сегментный регистр CS загружается селектор нового сегмента кода (а в теневой регистр загружается дескриптор этого сегмента), это и есть переключение на другой сегмент кода.

Вызов через шлюз является легальным способом обращения к системным (привилегированным) процедурам из задач обычного пользователя. При вызове через шлюз в самом дескрипторе шлюза находится селектор кодового сегмента, где находится процедура, и ее адрес в этом сегменте, таким образом, заданный в команде адрес *offset* игнорируется. Проверяется, что текущий уровень привилегий вызывающей программы CPL (из селектора в CS) *не меньше*, чем уровень привилегий самого шлюза вызова RPL (из селектора *seg*), но *не больше*, чем уровень привилегий вызываемой процедуры DPL (из дескриптора её сегмента кода).

Это "хитрое" условие означает, что программа может обращаться только к разрешённым ей селекторам шлюзов (CPL≤RPL), но все шлюзы не должны вести на менее привилегированный уровень (CPL≥DPL) (можно пользоваться только разрешёнными лифтами, но все они едут только вверх, к начальству 😊), или в крайнем случае "вбок", к коллегам, но никак не вниз, к подчинённым). Это, например, запрещает операционной системе вызывать процедуры обычного пользователя через шлюз.

Как уже говорилось, более привилегированная процедура обязана использовать свой собственный (системный) сегмент стека, поэтому для таких вызовов процессор *перемещает* из стека вызывающей программы в системный стек адрес возврата (значения CS и EIP). В дескрипторе шлюза вызова предусмотрено также поле Count длиной пять бит, его значение равно количеству двойных слов (**dd**), которое процессор *сам* копирует из стека вызывающей программы и помещает в системный стек вызываемой процедуры (вслед за регистрами EIP и CS). Это от 0 до 31 фактических параметров системной процедуры, их записывает в *свой* стек вызывающая программа. Сразу вслед за фактическими параметрами процессор помещает в системный стек значения регистров ESP и SS (дополненного до 32-х бит), что позволяет системной процедуре, при необходимости, "наведаться" в стек вызывающей программы, а также выполнить возврат с восстановлением стека пользователя (т.е. регистров ESP и SS).

Как будет ясно из дальнейшего описания вызова процедур на Ассемблере, фактические параметры передаются в дальнюю процедуру так же, как и в обычную (близкую) процедуру. В описании системной процедуры (системного вызова), естественно, описывается, какие параметры и сколько необходимо передать. Переход через шлюз на начало процедуры производится по правилу

CS:=селектор из дескриптора шлюза вызова;

EIP:=offset из дескриптора шлюза вызова

Таким образом, вызов через шлюз является не прямым, а косвенным вызовом процедуры (косвенным переходом).

И, наконец, вызов процедуры через дескриптор задачи TSS, приводит к так называемому переключению задач, этот механизм будет подробно описан в следующей главе, посвящённой системе прерываний.

Дальний возврат из процедуры по команде **retf** выполняется по схеме:

Изстека (EIP) ; Изстека (CS) ; $ESP := (ESP + i16) \bmod 2^{32}$

При выполнении этой команды, естественно, проверяются привилегии, так, для согласования с правилами вызова процедуры, невозможно вернуться на более высокий уровень привилегий (все лифты едут только вниз или "вбок"). При возврате на менее привилегированный уровень команда **retf** предварительно переключается на старый стек, используя значения регистров ESP и SS из системного стека, а также очищает сегментные регистры DS, ES, FS и GS, если их значения недействительны для более низкого уровня привилегий (иначе их использование будет вызывать сигнал прерывания).

На рис. 6.9. показан вид стека программы и системного стека перед командой дальнего возврата **retf** 8 (с двумя параметрами Param1 и Param2, т.е. Count=2) при вызове процедуры через шлюз. Индекс P у регистров обозначает вызывающую программу пользователя, а индекс S – системную процедуру.

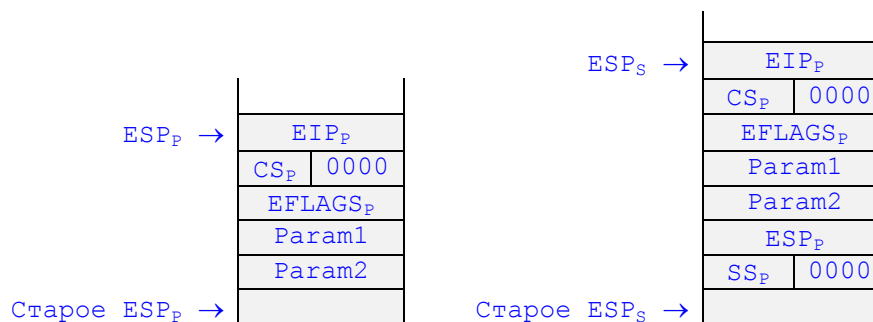


Рис. 6.9. Вид стека программы и системного стека перед командой возврата **retf** 8 (два параметра, Count=2) при вызове процедуры через шлюз.

Дополнительное действие команд возврата для параметра $i16 < > 0$

$ESP := (ESP + \text{Longword}(i16)) \bmod 2^{32}$

приводит к перемещению указателя вершины стека ESP. В большинстве случаев этот операнд имеет смысл использовать для $i16$ кратных четырём, и только тогда, когда $ESP + i16$ не выходит за нижнюю границу (дно) стека. В этом случае из стека удаляются $i16$ байт, что можно трактовать как *очистку* стека. Возможность очистки стека при выполнении команды возврата, как уже было показано ранее, является весьма полезной при программировании процедур на Ассемблере, она является аналогом уничтожения локальных переменных и фактических параметров процедур и функций Паскаля при выходе из блоков. Заметьте, что при переключении стеков $i16$ байт удаляются как из системного стека, так и из стека пользователя, т.е. задача пользователя не заметит, что работа велась не в её стеке, а в системном.

На этом заканчивается изучение процедур в языке Ассемблера.

Вопросы и упражнения

1. Какое значение имеет регистр ESP перед выполнением первой команды программы ?
2. Можно ли использовать в программе команду вызова процедуры **call** в виде `call eax` или `call ax` ?
3. Почему в языке машины не реализована команда `pop CS` ?
4. Чем понятие процедуры в Ассемблере отличается от понятия процедуры в Паскале ?
5. Обоснуйте необходимость принятия стандартных соглашений о связях между процедурой и вызывающей её программой.
6. Что такое прямой и обратный порядок передачи параметров в процедуру? Когда может возникнуть необходимость в обратном порядке передачи параметров в процедуру через стек ?

7. Что такое стековый кадр ?
8. Обоснуйте, какие из пунктов стандартных соглашений о связях **необходимы** для обеспечения *рекурсивных* вызовов процедур и функций.

ⁱ Кроме перечисленных в этой таблице допустимы также операнды формата `i16:i32 (i48)` и `m16:m32 (m48)`. Обычно такие операнды обозначаются как `seg:offset`, они задают межсегментный переход. Как производится такой переход будет рассказано при изучении команды вызова процедуры `call`. К сожалению, хотя в языке машины команда формата `jmp i48` есть (её код операции `0EAh`), но непосредственно записать эту команду на Ассемблере нельзя. Видимо, это сделано потому, что такая команда бесполезна в практическом программировании, так как при написании программы почти всегда не известно, в каком конкретно месте логической памяти будет находиться точка дальнего перехода и какой селектор имеет нужный кодовый сегмент. Как станет ясно далее при изучении темы модульного программирования, для дальнего прямого абсолютного перехода обычно используются так называемые статические связи между модулями по управлению с помощью внешних имен и входных точек, при этом команда формата `jmp i48` формируется Ассемблером автоматически из команды `jmp L`, где `L` – внешняя метка дальнего перехода. Чтобы непосредственно задать в программе команду `jmp i48` в нашем Ассемблере приходится пускаться на хитрости. Например, для дальнего перехода по адресу `seg:offset=5:12345678h` можно воспользоваться неразличимостью команд и данных и записать команду `jmp 5:12345678h` в виде набора констант: `db 0EAh, 78h, 56h, 34h, 12h, 05h, 00h` (учитывая, что адрес `i48` представляется в памяти в перевёрнутом виде). Более "цивилизованный" (и более длинный) обходный путь заключается в записи адреса `seg:offset` в стек и выполнения затем команды дальнего возврата:

```
push 12345678h
push word ptr 5
retf
```

ⁱⁱ Команду `SETcc` иногда используют оптимизирующие компиляторы, чтобы избавиться при вычислении условных операторов от команд условных переходов (как ни странно это звучит). Пусть, например, надо закодировать условный оператор (пусть `X` знаковое число)

```
if X<0 then Y:=200h else Y:=300h
```

Нижe показаны фрагменты на Ассемблере, полученные при программировании "в лоб" и оптимизирующим компилятором:

<pre>mov Y, 200h cmp X, 0 jl L add Y, 100h L:</pre>	<pre>xor ecx, ecx cmp X, 0 setl cl; if X<0 then cl:=1 else cl:=0 dec ecx; if X<0 then ecx:=0 else ecx:=-1 and ecx, 0FFFFFFC00h; if X<0 then ecx:=-100h else ecx:=0 add ecx, 300h mov Y, ecx</pre>
---	--

Вторая реализация, хотя и длиннее на 3 команды, но содержит только два обращения в память, вместо трёх в первом случае, но самое главное, в ней нет команд условного перехода. Как будет описано в 14 главе, это значительно ускоряет выполнение программы на конвейере.

ⁱⁱⁱ Логически вся оперативная память разделяется на так называемые страницы (обычно размером 4 Кб), причём страница памяти, расположенная сразу *после* дна стека, задаче никогда не выделяется, и при попытке доступа в неё возникает исключение "ошибка доступа в память" (`EXCEPTION_ACCESS_VIOLATION`). Чаще всего это возникает при исчерпании стека (`stack underflow`), т.е. при попытке чтения из пустого стека.

Страница памяти, расположенная сразу *перед* текущей вершиной стека (и ещё не принадлежащая ему), называется сторожевой страницей (`guard page`). При попытке доступа в сторожевую страницу фиксируется исключение "нарушение доступа к сторожевой странице" (`STATUS_GUARD_PAGE_VIOLATION`), при этом ОС Windows, обработав это событие, присоединяет сторожевую страницу к стеку (увеличивая его размер), назначая при этом новой сторожевой следующую страницу перед вершиной стека. Таким образом, при счёте задачи стек может расти в пределах отведённого ему максимального выделенного размера памяти (обычно 1 Мб, но может задаваться и большего размера при создании выполняемого файла).

При попытке доступа в *предпоследнюю* сторожевую страницу стека для задачи возникает исключение "переполнение стека" `EXCEPTION_STACK_OVERFLOW`. Две последние страницы стека являются резервными, их назначение в следующем. Задача пользователя может сама обработать ошибку "переполнение стека", реализовав для этого специальную процедуру (так называемый обработчик структурных исключений), которая будет вызываться ОС Windows при возникновении этого исключения. Вот для нормальной работы такой процедуры и предназначен этот "неприкосновенный запас" в виде двух последних страниц стека. И, наконец, перед последней резервной страницей, как и после стека, обязательно располагается "чужая" страница, попытка доступа в которую, как уже говорилось, вызывает исключение `EXCEPTION_ACCESS_VIOLATION`.

Таков вкратце механизм работы со стеком в ОС Windows. Таким образом, по умолчанию в начале счёта под стек отведены всего три страницы, одна "настоящая", одна сторожевая и одна резервная. Заметим также, что после роста, снова уменьшатся в размерах стек не может. Изложенный выше механизм работы задачи со

стеком существенно усложняется, если задача запускает несколько вычислительных потоком (нитей – threads). Все потоки задачи разделяют общие секции кода и данных, но каждый поток должен иметь свой стек. Более подробно об этом говорится в главе, посвящённой мультипрограммному режиму работы ЭВМ.

Забегая вперёд отметим, что, в отличие от стека задачи, под так называемый *системный* стек, о котором будет говориться в следующей главе, отводится всего три страницы (четыре в 64-битных ЭВМ), т.е. он совсем маленький и динамически расти не может.

Впервые растущий стек со сторожевыми страницами был реализован ещё в операционной системе древней ЭВМ PDP фирмы DEC (Digital Equipment Corporation) в 70-х годах прошлого века.

^{iv} Не надо понимать высший уровень привилегий слишком буквально, в том смысле, что программе, работающей на этом уровне "всё можно". Как видно из рис. 6.8, программе, работающей на нулевом уровне привилегий, *запрещено* передавать управление в непривилегированную программу на третьем уровне привилегий, как и использовать непривилегированный стек. Легко понять причину такого запрета, так как программа пользователя может содержать ненадёжный код, в котором, например, из-за выхода индекса за границы массива могут быть испорчены важные данные, скажем, таблица дескрипторов. Это, конечно, не значит, что привилегированная программа вообще не может выполнять код из менее привилегированной, просто сначала она должна изменить дескриптор сегмента кода непривилегированной программы, подняв его уровень до своего. Это значит, что привилегированная программа *осознанно* идёт на риск выполнения ненадёжного кода.