Глава 6. Язык Ассемблера

Ассемблер – это удивительный язык, открывающий дверь в мир больших возможностей и неограниченного самовыражения.

Крис Касперски ака мыщъх

6.1. Понятие о языке Ассемблера

Системное программирование хранит множество секретов, загадок и тайн, постепенно становясь уделом небольшой горстки профессионалов, в то время как мир дружно сходит с ума, подсаживаясь на языки высокого уровня, которые чем дальше – тем всё выше и выше. Об ассемблере вспоминают только тогда, когда требуется что-то очень сильно нестандартное, с чем компилятор уже не справляется или сгенерированный им код не отвечает требованиям производительности.

Крис Касперски ака мыщъх

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

Впрочем, для некоторых машинных команд в языке Ассемблера может быть не предусмотрено соответствующих им мнемонических обозначений кодов операций. Обычно это такие команды, которые самому 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 операнда с кодом операции `66h`, префиксы задания векторных команд VEX, VEX2 и VEX3 и префикс задания 64-битных регистров REX, для этих команд в нашем Ассемблере нет мнемонических обозначений. Программист может вставлять такие команды в свою программу только в виде констант, пользуясь принципом фон Неймана неразличимости команд и чисел.

Принято считать, что термин «Assembler» впервые стал использовать английский учёный Морис Уилкс (Maurice Wilkes) в поздних описаниях «полностью электронной» ЭВМ EDSAC где-то в 1950 году. Правда, там под этим термином понималось просто объединение полей машинной команды в некоторое «командное слово». Целью было представление машинной команды в виде, более подходящем для чтения человеком.

Для перевода с языка Ассемблера на язык машины используется специальная программа-переводчик (транслятор), также называемая *Ассемблером* (от английского слова «assembler» – «сборщик»).[1] 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.

---

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

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 $\rightarrow 12_{10} \rightarrow 01100_2$);
- 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 это адрес переменной – смещение от начала памяти (или начала некоторого сегмента, что будет рассмотрено далее);
- преобразовывать числа, написанные в программе в различных системах счисления во внутреннее машинное представление (в машинную систему счисления).

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, эти принципы выполняются в конкретном компьютере):

- возможность помещать в любое определённое программистом место памяти своей программы любую команду или любые данные (даже если эти области памяти закрыты на запись);
- возможность выполнять любые данные как команды и работать с командами, как с данными (например, складывать команды как числа);
- возможность задать в своей программе и выполнить любую команду из языка машины.

Как уже упоминалось в разд. 5.7, современные ЭВМ могут работать в так называемом привилегированном (или защищённом) режиме. 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. Более полно привилегированный режим работы и защита памяти будет изучаться в главе, посвящённой мультипрограммному режиму работы ЭВМ.

6.2. Применение языка Ассемблера

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

Народная мудрость

If you want to write the best high-level language code, *learn assembly language.*

Common programming advice

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, приходится тратить примерно 10 предложений на Ассемблере. Кроме того, понимать, отлаживать и модифицировать программы на Ассемблере значительно труднее, чем программы на языках высокого уровня.

Начинать учиться программированию надо не с Ассемблера, а с одного из языков 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.

Отметим, что при трансляции с некоторых языков высокого уровня получаются не программы на языке машины и даже не объектные модули, а программы на так называемых промежуточных языках или в промежуточном представлении IR (Intermediate Representation). Эти промежуточные языки обычно называются байт-кодами, Псевдоассемблерами, Web Assembly, LLVM (Low Level Virtual Machine) и т.д., по своему уровню они близки к языку Ассемблера. В дальнейшем программы с этих промежуточных языков либо компилируются на язык объектных модулей (реже на язык машины), либо выполняются интерпретатором. Отметим также, что многие компиляторы (в частности, с языков C, C++ и Free Pascal) в качестве промежуточного результата могут выдавать текст и на «настоящем» Ассемблере.

Некоторые промежуточные языки (например, LLVM) являются машинно-независимыми, а уже с них создаются оптимизирующие компиляторы на конкретную архитектуру ЭВМ.

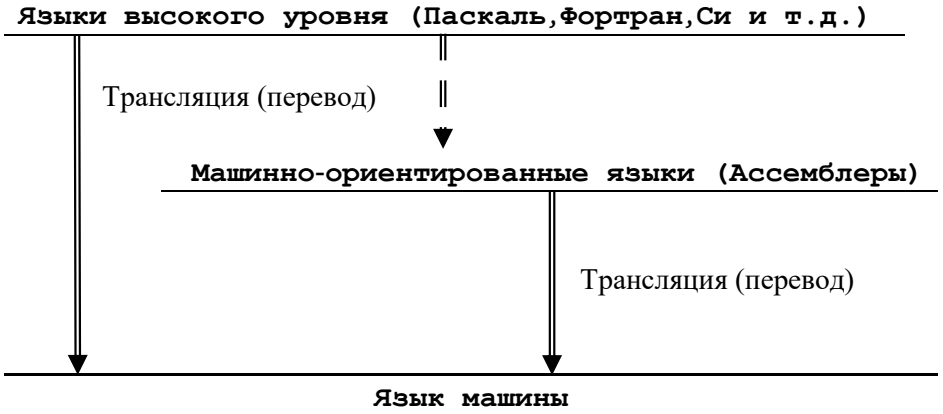


Рис. 6.1. Взаимосвязь языков программирования разных уровней.

Естественно, возникает вопрос, когда у программиста может появиться необходимость при написании 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.

Вторая область применения Ассемблера связана с оптимизацией выполнения тех *больших* программ, которые требуют много времени для своего счёта.

Обычно в программистской литературе *большими* (по размеру) принято называть программы, содержащие порядка 100 тысяч и более строк исходного текста. Например, оптимизирующий компилятор Clang для языка C++ содержит примерно 2 млн. строк исходного кода.

Кроме того, большими называются и программы, требующие много (сотни, тысячи и миллионы) часов машинного времени для своего выполнения. В качестве курьёзного примера такой программы можно привести решение в целых числах уравнения $X^3+Y^3+Z^3=K$. Доказано, что для K, которые при делении на 9 дают в остатке 4 или 5, решение невозможно. К 2019 году почти все решения этого уравнения для остальных чисел K от 1 до 100 уже были найдены, число $K=42$ оставалось последним.

Для решения задач, требующих большого времени счёта, часто используются так называемые GRID-системы. Эти системы, возникшие в конце 90-х годов прошлого века, использует свободные вычислительные ресурса компьютеров в сети Интернет. Во время простоя персонального компьютера, вместо выполнения программы-заставки или перехода в спящий режим, можно считать вариант программы, присланной по сети GRID-системой. Отметим, что в некоторых организациях не принято отключать персональные компьютеры на ночь, так как считается что выключение и включение машины наносит ей больше вреда, чем непрерывная работа. Таких GRID-систем много, можно отметить SETI@home для поиска внеземных цивилизаций, Fusion для разработки термоядерного реактора, LHC Computing Grid для обработки данных с Большого адронного коллайдера, Einstein@Home для проверка гипотезы Эйнштейна о гравитационных волнах и поиска радио- и гамма-пульсаров, Rosetta@Home для исследования разработки лекарств, Collatz Conjecture для проверки гипотезы Коллатца и другие. Существует даже стандарт на построение таких систем OGSA (Open Grid Services Architecture). Получающийся «виртуальный» компьютер по мощности сопоставим с супер-ЭВМ.

На большой GRID-системе международной организации Charity Engine решение уравнения $X^3+Y^3+Z^3=42$ заняло около миллиона часов времени (в пересчёте на один компьютер). В отдельные

моменты времени параллельно работали более 100000 персональных ЭВМ, а всего было задействовано 500000 машин. В итоге было найдено решение:

$$-80538738812075974^3+80435758145817515^3+12602123297335631^3=42$$

Разумеется, никакой уверенности, что это единственное решение нет. Например, для `K=3` давно были известны два решение $1^3+1^3+1^3=3$ и $4^3+4^3+(-5)^3=3$, а совсем недавно найдено третье решение

$$569936821221962380720^3-569936821113563493509^3-472715493453327032^3=3$$

На поиск этого решения в системе Charity Engine потребовалось уже 4 миллиона часов времени (в пересчёте на один персональный компьютер). При увеличении диапазона чисел `K` от 100 до 1000 пока остались нерешёнными уравнения для чисел 114, 390, 627, 633, 732, 921 и 975. Конечно, теорема о том, что такие решения обязательно существуют, не доказана.

Другим забавным использованием Grid-систем является проверка так называемой гипотезы Коллатца. В 1932 году немецкий математик Лотар Коллатц сформулировал гипотезу, что для любого `N>0` итерационный процесс

```
if Odd(N) then N:=3*N+1 else N:=N div 2
```

всегда сходится к единице.[1] Отметим, что это частный случай задачи остановки алгоритма, в общем виде эта задача является алгоритмически неразрешимой проблемой. Теорема Коллатца полностью не доказана, пока только показано, что эта теорема верна «почти для всех чисел», однако к 2021 году уже проверены все N до 9 789 690 303 392 599 179 036. Чем только не занимаются люди… 🙂

Часто компиляторы 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 компилятор. В основном это касается программ вычислительного характера, которые бóльшую часть времени (более 99%) выполняют 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, 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.

Ещё одна область применения Ассемблера – написание программ, от которых требуется предельная эффективность по времени и/или по занимаемой памяти. Это, например, библиотеки стандартных программ. Стоит отметить, что есть и «специфические» области применения Ассемблера, например, реверс-инжиниринг (reverse engineering). Под этим обычно подразумевают исследование и преобразование программ на машинном языке (исходных текстов которых нет) в программы на языках более высокого уровня (в основном на Ассемблер), в целях их изучения и изменения. Обычно для этого применяют так называемый *дизассемблер*, программу, которая преобразует текст на машинной языке в программу на Ассемблере.

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 компиляторами 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.

Итак, хорошие программисты знают Ассемблер, но почти никогда не пишут на нём. Они просто хорошо представляют, во что превратится написанная ими конструкция на языке высокого уровня: или в несколько простых команд, или в хитросплетение вложенных циклов и условных переходов.

[1] У этой гипотезы есть много других названий, например: «дилемма 3n+1», «гипотеза градины», «сиракузская проблема» и другие.

6.3. Структура программы на Ассемблере

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

Алан Перлис,
первый лауреат премии Тьюринга

Единственный способ изучать новый язык программирования – писать на нем программы.

Брайан Керниган

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), MASM-6.14. Это одна из последних версий этого Ассемблера, ориентированная *только* на 32-битную архитектуру, а версии, начиная с MASM-7.XX и далее поддерживают уже 64-битную архитектуру.

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.[1] В каком смысле модуль является *независимой* единицей языка Ассемблер, будет объяснено несколько позже, когда в Главе 9 будет изучаться тема «Модульное программирование». Наши первые программы будут содержать всего один модуль,[2] но позже будут рассмотрены и простые двухмодульные программы.

Каждый модуль на Ассемблере обычно содержит описание одной или нескольких секций для размещения в памяти команд и данных. 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 разместить команду среди данных. Исключением является размещение в секции команд *констант*, это делают многие из современных компиляторов с языков высокого уровня. Таким образом, 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, можно использовать и отдельные секции констант (закрытых на запись). 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).

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

---

[1] Вместе с модулями, программа может использовать и другие функционально-независимые части, например, такие, как "пакет прикладных программ", "библиотека стандартных программ", "библиотека классов" и др.

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

6.4. Классификация предложений языка Ассемблер

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

Алан Перлис,
первый лауреат премии Тьюринга

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

- Комментарии.

Комментарии в программе должны быть похожими на кружевные трусики: маленькими, прозрачными, и оставляющими достаточно места для воображения.

Марк Цукерберг

Код никогда не лжёт, а вот с комментариями такое случается.

Рон Джеффрис
Основатель методологии
экстремального программирование 🙄

Комментарии не влияют на вид полученной при компиляции машинной программы, они предназначены исключительно для чтения человеком. 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 его неким частным типом предложения, хотя не все учебники по Ассемблеру придерживаются этой точки зрения. Способ записи многострочных комментариев:

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

здесь служебное имя **`comment`** определяет начало многострочного комментария, а <символ-ограничитель> задаёт его границы, он похож на символы начала и конца комментария **`{`** и **`}`** в языке Паскаль, однако этот символ не должен встречаться внутри самого комментария, например:

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

- Команды.

Не суетитесь, командовать парадом буду я!
И.Ильф и Е.Петров. «Золотой телёнок»

Почти каждому предложению языка Ассемблера этого типа будет соответствовать одна команда (instruction) на языке машины. Иногда из одного предложения получаются несколько «тесно связанных» команд, обычно говорят, что перед основной командой ставятся одна или несколько так называемых команд-префиксов.

- Резервирование памяти.

Любая программа стремится занять всю доступную память.

14-й закон программирования

В той секции, где записаны эти предложения, резервируются области памяти для хранения переменных. Это некоторый аналог раздела описания переменных **`var`** в языке Паскаль. Заметим, что во многих учебниках такие предложения называют *директивами* резервирования памяти. Полные пра-

вила записи этих предложений можно посмотреть в учебниках [5-7], здесь приведены лишь некоторые примеры с комментариями.

Предложение	Количество байт памяти
`A db ?`	1 байт (define byte)
`B dw 1`	2 байта (слово) (define word)
`C dd ?`	4 байта (двойное слово) (define double)
`dd 7`	*безымянная* область длиной 4 байта
`D df 0`	6 байт (define far)
`E dq -1`	8 байт (define quad words)
`F dt ?`	10 байт (define ten bytes)
`G oword ?`	16 байт (define octa words)

В этих примерах описаны именованные и безымянные области памяти разной длины. Именованные области можно (в определённом смысле) считать аналогами *переменных* в языках высокого уровня. В нашем примере переменные с именами `B`, `C`, `D` и `E` имеют начальные значения соответственно `1`, `7`, `0` и `-1`, а остальные переменные, как и в стандарте языка Паскаль, не будут иметь конкретных начальных значений, что отмечено символом вопросительного знака в поле параметров.

Ассемблер MASM позволяет в качестве служебного слова **db** использовать синонимы **byte** и **sbyte**, вместо **dw** можно использовать синонимы **word**, **integer** 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**	**integer → integer**
sword → smallint	**dword → longword**	**sdword → longint**	**tbyte → extended**

В Ассемблере, однако, знаковые и беззнаковые переменные различаются не в обычных (машинных) командах, а только в макросредствах, о чём будет говориться в Главе 11.

Итак, пусть, например, у нас есть такое резервирование памяти:

```
A db ?
B dw 1234h
C dd 12345678h
D db 23q;    23₈=19₁₀
E db 1001b; 1001₂=9₁₀
  dd 10;     10₁₀
```

На языке Free Pascal переменные `A`, `B`, `C`, `D` и `E` можно, например, описать так:

```
var A: byte;
    B: word=$1234;{B:34h; B+1:12h}
    C: Longword=$12345678;
{ C:78h; C+1:56h; C+2:34h; C+3:12h}
    D: byte=&23; 8-ная система счисления
    E: byte=%1001; 2-ная система счисления
{ dd 10h ??? безымянных переменных нет }
```

По принципу фон Неймана, однако, ничего не мешает нам на работать напрямую с одним или несколькими байтами, расположенными подряд в любом месте памяти. Например, команда

```
mov ax,B+1; ax=7812h=C:78h; B+1:12h;
```

будет читать на регистр `AX` слово, *второй* байт которого располагается в конце переменной `B`, а *первый* – в начале переменной `C` (надо помнить о «перевёрнутом» хранении слов в памяти!). А команда

```
mov eax,C+4; eax=C+4: 10h 1001b 23q 12h;
```

будет читать на регистр `EAX` “безымянную” переменную, начинающуюся с адреса `C+4`., причём из-за перевёрнутого представления в памяти, байт `12h` из адреса `C+4` попадёт в младшую часть регистра `EAX`, т.е. в регистр `AL`. Поэтому следует быть осторожными и не считать `A`, `B`, `C` и т.д. отдельными, «независимыми» переменными в смысле языка Паскаль, это просто именованные области памяти. На самом деле пример на языке Free Pascal приведён только для иллюстрации, он не совсем корректен [i] [см. сноску в конце главы]. 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

```
M db 20 dup (?)
; M: array[0..19] of {byte,char,shotint,boolean...}
```

С помощью операции-дубликатора **dup** резервируется 20 подряд расположенных байт с неопределёнными начальными значениями.[1] Это можно назвать резервированием памяти под массив из 20 элементов, причём значение имени `M` является адресом первого из этих элементов.

Отметим, что в нашем Ассемблере MASM дубликатор выглядит «прилично», а, например, в Ассемблере NASM это будет выглядеть малопонятно и по разному:

```
M: times 20 db 1; с начальным значением 1
M: resb  20;      без начального значения
```

Отметим, что так же, как в Ассемблере, понимается массив и в языке С:

```
unsigned char M[20];
```

При этом в языке С оператор присваивания `M[2]=1` (третий элемент массива `:=1`) эквивалентен `M+2=1`, `[M+2]=1` и даже `2[M]=1`. Аналогично на Ассемблере можно писать команды для такого же присваивания:

```
mov M[2],1
mov M+2,1
mov [M+2],1
mov [2+M],1
mov 2[M],1
mov [M]2,1; К счастью, ошибка
```

Понятно, почему язык С называют *машинно-ориентированным* языком программирования. Количество элементов массива можно задать и нулевым, это будет область памяти нулевой длины:

```
M1 dw 0 dup (?)
; M1 dw 10 dup (?)
M2 db 20 dup (?)
   db 0 dup (?); ???
```

В этом случае у массива `M2` из 20 элементов-байт будет дополнительное имя `M1`, которое программист может, например, считать именем массива из 10 слов (**dw**), что иногда удобно. В следующем разделе говорится, как для этого же программист может использовать «пустую» директиву `M1 label word`.

- Директивы.

Усердный в службе не должен бояться своего незнанья; ибо каждое новое дело он прочтёт.
Козьма Прутков

В некоторых учебниках директивы называют *командами Ассемблеру* (т.е. *компилятору* с Ассемблера, а иногда псевдокомандами), что хорошо отражает их назначение в программе. Эти предложения, за редким исключением не порождают в машинной программе никаких команд или переменных. Директивы используются программистом для того, чтобы давать компилятору Ассемблера определённые указания, задавать синтаксические конструкции (например, типы данных) и управлять его работой 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 `Very_Long_Name_of_Next_Statement` является именем следующего за ней предложения, таким образом, у этого предложения будет две метки (два имени):

```
Very_Long_Name_of_Next_Statement:
L: mov eax,2
```

---

[1] В отличие от скаляной переменной, например, `X db ?` значение дублируемой переменной всегда заключается в круглые скобки `X db 10 dup (?)`.

Аналогичную «пустую» метку можно поставить и в секциях констант и данных, правда, надо явно задать тип области памяти, на которую указывает эта метка (иначе компилятор не знает тип этой метки):

```
A label word
B equ this word
C dw 0 dup (?)
D dw ?
```

Теперь метки A, B, C и D задают один и тот же адрес в секции данных.

- Макрокоманды.

Не грусти. Рано или поздно всё станет понятно, всё станет на свои места и выстроится в единую красивую схему, как кружева. Станет понятно, зачем всё было нужно, потому что всё будет правильно.
Льюис Кэрролл.
«Алиса в стране чудес»

Этот класс предложений Ассемблера относится к *макросредствам* языка, и будет подробно изучаться в 11 главе. Пока надо лишь сказать, что на место макрокоманды при трансляции в программу по определённым правилам подставляется некоторый набор (возможно и пустой) предложений Ассемблера.

6.4.1. Структура предложения Ассемблера

Тебе что-то непонятно? Перечитай и перепиши несколько раз. Сначала непонятное станет привычным, а затем привычное – понятным.
Студенческая мудрость

За редкими исключениями, каждое предложение Ассемблера 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, 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 должно быть *командой*. Такое предложение должно располагаться в секции команд и на него можно, как говорят, *передавать управление*.

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 в Главе 11):

```
K = K+1
```

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

```
My_Statment_Name: mov eax,  \
```

```
My_Longint_Variable_Name \
[eax+8*ebx];  предложение в три строки
```

Кроме того, перенос на новую строку можно делать просто по символу запятой при описании переменных с начальными значениями, например:

```
X1_10 dd 1,2,3,4,5,;  первые пять элементов массива
6,7,8,9,10;  вторые пять элементов массива
```

Отметим, что однострочные *комментарий* и строки в кавычках (quote) и апострофах (apostrophe) 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 одного предложения 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.), имена регистров (EAX, EBX и т.д.), имена директив (**proc**, **macro**, **end**, **.code** и др.). Как и в Паскале, будем для читабельности выделять служебные имена (кроме имён регистров) **жирным** шрифтом. 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**, **Str** и другие.

Как и в Паскале, в языке Ассемблера принято такое же соглашение по семантике всех *служебных* имён: большие и маленькие буквы в них не различается. Таким образом, служебные 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 Ассемблера, что эти имена чувствительны к регистру символов, так что `My_Name` и `My_name` – это *разные* имена.

Этот режим различения регистра устанавливается директивой **`option casemap:none`**, без неё, как и в Паскале, большие и маленькие буквы во всех именах (пользователя) не будут различаться. К сожалению, нам приходится вставлять эту директиву в наши программы на Ассемблере, так как они почти всегда используют подпрограммы из системных библиотек (написанные на языке C), в которых считается, что большие и маленькие буквы имён подпрограмм *различаются* 🙁.

По умолчанию с точки могут начинаться только *служебные* имена, однако тем программистам, которым почему-то ну очень нужно начинать с точки свои собственные имена, могут сделать это, задав опцию компилятора **`option DOTNAME`**.

Итак, служебные имена (скажем код операции **sub**) нельзя использовать в качестве имени пользователя. Однако, как это часто бывает, когда чего-то «нельзя, но оч-е-е-нь хочется», Ассемблер допускает исключить некоторые слова из списка служебных имён, это делается с помощью директивы (опции) компилятора с именем **`NOKEYWORD`**, например:

```
    div eax
    option NOKEYWORD:<div inc>
div dd  ? ; Теперь div – это обычная метка
    inc eax; ОШИБКА, не описано имя inc!
```

Данная опция действует от места её объявления вниз по тексту программы. Можно рекомендовать исключить из служебных имён такие часто используемые программистами имена, как **C**, **Name** и **Str**:

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

Разумеется, исключения из списка ключевых таких имён, как **.code** или, тем более **end**, приводит к ошибкам компиляции «чуть чаще, чем всегда». Для наших программ директива исключения

некоторых служебных имён будет автоматически вставляется в текст программы на Ассемблере при использовании предложения `include console.inc`., о чём будем говорить далее.

6.5. Пример полной программы на Ассемблере

Преподавание программирования – дело почти безнадёжное, а его изучение – непосильный труд.

Чарльз Уэзерелл.
«Этюды для программистов»

Сам по себе ассемблер не обеспечивает ни компактности кода, ни высокой скорости. Всё решают хитрые трюки и приёмы программирования, находчивость и инженерная смекалка, наконец!

Крис Касперски ака мыщъх

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, 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 такой макрокоманды Ассемблер будет при компиляции подставлять соответствующий этой макрокоманде набор предложений языка Ассемблер (так называемое *макрорасширение*). 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. Консольное приложения

Нужно думать не о том, что нам может пригодиться, а о том, без чего мы не сможем обойтись.

Джером К. Джером.
«Трое в лодке, не считая собаки»

При запуске программы в ОС Windows в текстовом (не графическом) режиме ей для работы предоставляется так называемая консоль. Консолью называется комплект устройств текстового интерактивного ввода-вывода (текстовый экран или окно на таком экране, физическая или экранная клавиатура, и, возможно, мышь или сенсорный указатель) для взаимодействия программы с внешним объектом (предположительно, человеком ☺).

Ассемблерные программы, описанные в этой книге, будут запускаться в окне консоли, эти программы называются консольными приложениями Windows. В отличие от графических окон, программа в Windows может иметь только одну консоль.

Перекодировка текста. 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 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, выпол-

---

[1] В машинном языке есть только команды для обмена одним байтом, словом, двойным (или четверным для 64-битных ЭВМ) словом между регистром процессора и заданным в команде особым периферийным устройством компьютера (портом ввода/вывода), см. разд. 14.3.1.

нив макрокоманду **ConsoleMode**. Повторное выполнение этой же макрокоманды снова включает режим перекодировки при выводе.

Это же относится и к *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 приходиться «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.

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 некоторый навык программирования на Паскале). Такие средства предоставляет набор макрокоманд с мнемоническими именами, похожими на имена из языка Free Pascal, они реализуют многие стандартные средства ввода/вывода и управление программой.[1]

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**.

> При описании макрокоманд будут использоваться фрагменты программ на Ассемблере, если что-то при этом Вам будет в них непонятно, то надо не обращать на это внимания и вернуться в этот раздел после полного изучения данной главы.

В программах на Ассемблере нам будут полезны следующие макрокоманды.

- **Макрокоманда очистки окна консоли**

  ClrScr

Она эквивалентна вызову процедуры **ClrScr** языка Free Pascal. Имеет синонимы **CLRSCR**, **clrscr** и **Cls**.

- **Макрокоманда вывода 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 (как и отдельный символ) может заключаться как в двойные, так и в одинарные кавычки (апострофы). Оператор Ассемблера **offset** <имя переменной> вычисляет адрес переменной в памяти. Потом об операциях **ptr** и **offset** будет говориться подробно. При задании *адреса* текста в виде **offset** T сам текст T в памяти должен заканчиваться нулевым байтом (признаком конца текста), 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. К сожалению, по умолчанию текст не центрируется в заголовке окна, а выравнивается по *левому* краю, поэтому рекомендуется ставить в начале текста несколько пробелов.

---

[1] Пакет нужных нам макрокоманд собран в один текстовый файл с именем io.inc, этот файл содержится (вместе с системой программирования MASM и примерами программ на Ассемблере) в дистрибутиве https://arch32.cs.msu.ru/semestr2/masm 6.14.zip.

- **Макрокоманда вывода строки текста**

  outstr[**ln**] text

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; Текст: Hello,"World"!,0
.code
   outstrln "Привет, мир!"
   outstr   offset T; Вывод Hello,"World"!
   mov      eax,offset T
   outstr   eax;      Вывод Hello,"World"!
   outstrln 'Don"t do that!'
   lea      eax,T[7]
   outstrln eax;      Здесь будет вывод "World" !
```

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** (или **Ln**) приводит к переходу после вывода на начало новой строки. Эта макрокоманда имеет синонимы **OutStr** и **OUTSTR**.

- **Макрокоманда ввода строки текста**

  inputstr buf,Len[,text]

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

В буфер помещается текст из стандартного потока stdin [1] длиной не более Len или до конца введённой строки (без символов конца строки #13 и #10). Текст в буфере будет заканчиваться нулевым символом, поэтому длина буфера должна быть по крайней мере на единицу больше Len. Действие этой макрокоманды похоже на ввод в языке Free Pascal строки S: **var** S:string[buf-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="
```

[1] 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 консоли, хотя возможны и другие подключения. Эти же соглашения действуют и в Ассемблере, хотя за стандартными потоками здесь, следуя языку C, закреплены имена stdin, stdout и stderr (последний поток предназначен для вывода сообщений об ошибках, обычно он тоже подключён к экрану).

Макрокоманда **outwordln** описывается далее. 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**

  inchar op1[,text]

где операнд op1 может иметь формат r8 или m8. Код (номер в алфавите) введённого символа записывается в место памяти, определяемое операндом op1. Эта макрокоманда эквивалентна последовательному выполнению операторов языка Free Pascal для вывода текста и ввода одного символа [Write(text);]Read(op1). Формат текста-приглашения такой же, как и в макрокоманде **outstr**. Макрокоманда имеет синонимы **Inchar**, **InChar** и **INCHAR**.

В реализации этой макрокоманды при переходе на следующую строку будет введён один символ LF (#10), а символы из дополнительного алфавита языка Free Pascal предваряются символом #0. 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]
```

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
```

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 (подробно о макрофункциях мы будем говорить при описании макросредств Ассемблера в главе 11). Макрофункция **@DOS** подставляет регистр AL вместо своего вызова, команда **cmp** x,**@DOS**('Ж') будет заменена на команду **cmp** x,AL.

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

```
  cmp @inchar("Введите цифру или точку = "),'.'
; будет заменяться на предложения
; inchar al,"Введите цифру или точку = "
; cmp al,'.'
  je  Tochka
  mov bl,@inchar()
```

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

Ограничение. Выводится фатальная диагностика при попытке ввода в регистр-указатель вершины стека `inchar [e]sp`.

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

  Readkey [text]

Код (номер в алфавите) введённого 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 (';') – это код клавиши **F1** в дополнительном алфавите. Имя макрокоманды имеет синонимы **readkey** и **READKEY**.

Наряду с *макропроцедурой* **Readkey** реализована и *макрофункция* `@Readkey([text])`, которая подставляет регистр AL, содержащий введённый символ, на место своего вызова, например:

```
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**. Имеет синонимы **OutChar**, **OUTCHAR**, **outch** и **OUTCH**.

Для задания кода символа удобно использовать символьную константу языка Ассемблер (например, 'A' или эквивалентную запись "A"), тогда можно не задаваться вопросом о соответствие 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([text,]op1)`. Как и в Паскале, добавление к имени макрокоманды окончания **ln** (или **Ln**) приводит к переходу после вывода на начало новой строки.

Следует внимательно относиться в *выводу* русских букв, *введённых* 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 = "
   outstr "Введена буква = "
   ConsoleMode ; Отключить перекодировку в DOS
   outchar X   ; Вывести РУССКУЮ букву
   ConsoleMode ; Включить перекодировку в DOS
```

Ограничение. К сожалению, пока макрокоманда работает неправильно, при использовании регистра ESP как базового, например `outchar byte ptr [esp]`, такой вывод надо делать в две команды, например, `mov al,[esp]` и `outchar al`.

- **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, фор-

мат этого последнего параметра такой же, как и в макрокоманде **outstr**. С точки зрение Паскаля макрокоманда **inint** выполняется как два оператора `[write(text);] read(op1)`. Макрокоманда имеет синонимы **Inint**, **InInt** и **ININT**.

Макрокоманда вводит из стандартного потока на место первого операнда любое целое значение из диапазона $-2^{31}..+2^{32}-1$ (это объединение диапазонов знаковых и беззнаковых чисел формата двойного слова **dd**). При выходе вводимого значения за максимальный диапазон двойного слова макрокоманда выдаёт во время счёта (предупредительную) диагностику об ошибке:

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

при этом операнду присваивается значение `7FFFFFFFh` (`MaxLongint`). Такой же результат будет и при явно заданном знаке минус и абсолютном значении введённого числа больше, чем `80000000h` (`MaxLongword`).

Флаг `CF:=0` при правильном вводе и `CF:=1` при неправильном.[1] Отметим, что это совпадает с поведение программы на языке Free Pascal в режиме ввода без контроля `{$I-}`, при этом `CF:=1`, если функция `IOResult` возвращает значение, не равное нулю. 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 при возврате устанавливается флаг `SF:=1`, иначе `SF:=0`. Например, из строки `"-123ABCD"` введётся число `-123`, во входном потоке останется строка `"BCD"`, при этом будет `CF:=1`, `SF:=1` и `ZF:=1`.

В случае, если введённое число превышает размер операнда `op1`, то производится *усечение* этого числа без выдачи диагностики об ошибке. Это эквивалентно работе программы на языке Free Pascal в режиме `{$R-}`, т.е. без контроля выхода величины за допустимый диапазон. Заметим, что Free Pascal по умолчанию работает именно в этом режиме. При необходимости в программах на Ассемблере контроль правильности ввода надо проводить самому программисту, вводя число в переменную формата **dd** и затем проверяя нужный диапазон.

Добавление к имени макрокоманды окончания **ln** (или **Ln**) приводит к очистке буфера ввода, это эквивалентно вызову стандартной процедуры `Readln` языка Паскаль, например:

```
.data
X  dd  ?
.code
inint X,"Введите целое число X="
  jc    BigNum; Число вне допустимого диапазона
  jz    Minus;  Введено число со знаком минус
  js    BadEnd; Плохое окончание лексемы числа
```

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

```
cmp @inint(),0; Это предложения:
; inint eax
; cmp eax,0
  je  ZeroInput
  outintln @inint("Введите число=");, \
                  "Введено число="
; inint eax,»Введите число=»
; outintln eax,,»Введено число=»
```

[1] Для любознательных. Так же реагирует системная функция `read` в операционной системе FreeBSD.

Ограничения. Выводится фатальная диагностика при попытке ввода в регистр-указатель вершины стека `inint [e]sp`.

- **Макрокоманды вывода целого числа**

  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**. Имеет синонимы **Outint**, **OutInt**, **OUTINT**, **OutI**, **outi**, **OutI**, **OUTI** и, соответственно, **Outword**, **OutWord**, **OUTWORD**, **OutU**, **outu**, **OutU**, **OUTU** (**U** в смысле Unsign).

Действие макрокоманды `outint op1[,[op2][,text]]` эквивалентно выполнению вывода одного знакового целого значения в языке Паскаль `Write([text,]op1[:op2])`, где параметр op2 задаёт ширину поля вывода. В отличие от Паскаля можно задавать отрицательное значение, при `op2<0` в качестве ширины поля вывода берётся абсолютная величина op2, но выводимые данные выравниваются не по правому, а по *левому* краю поля вывода.

Действие же макрокоманды с именем **outword** отличается только тем, что первый операнд при выводе *трактуется* как беззнаковое (неотрицательное) целое число. При невозможности определить тип выводимого значения предполагается **dd**, например, `outint [ebx]` выполняется как `outint dword ptr [ebx]`.

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,10,"Введено x="
  outwodln esp,,"Адрес вершины стека="
  outu [esp],,"Адрес возврата="
  OutU -1,-12,"MaxLongWord="
```

Примечание. Макрокоманды **outint** и **outword** используют в своей работе вспомогательную макрокоманду с именем **outnum**, которая умеет выводить (беззнаковые) целые числа в двоичной и шестнадцатеричной системе счисления (см. файл с текстами макроопределений io.inc), например:

```
  mov al,-1
  outnum al,,b; Вывод 11111111b
  outnum al,,x; Вывод FFh
```

- **Макрокоманда перехода на новую строку**

  newline [n]

Эта макрокоманда предназначена для перевода курсора к началу следующей строки экрана консоли и эквивалентна вызову n раз стандартной процедуры без параметров Writeln языка Паскаль. По умолчанию `n=1`. Имеет синонимы **NewLine**, **Newline** и **NEWLINE**.

- **Макрокоманда без параметров**

  flush

Эта макрокоманда 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 концевого служебного символа LF, уже введённого, например, по **inchar**). Имеет синонимы **Flush** и **FLUSH**.

- **Макрокоманда перемещения курсора**

  GotoXY x,y

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

- **Макрокоманда получения X-позиции курсора**

  WhereX

Номер столбца позиции курсора (в обычном режиме 1..80) в окне консоли возвращается в регистре AL. Эта макрокоманда эквивалентна вызову функции `AL:=WhereX` языка Free Pascal. Имеет синонимы **wherex** и **whereX**.

- **Макрокоманда получения 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. Имеет синонимы **wherey** и **whereY**.

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]

Имеет синоним **Settextattr**. 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 графическое окно сообщения с текстом заголовка Caption, текстом сообщения Message и кнопками ответов. Имеет синонимы **msgbox** и **MSGBOX**. Пиктограмма окна сообщений, количество и вид кнопок определяется стилем окна Style. Стиль окна определяется набором (суммой) битовых признаков, которым присвоены следующие мнемонические имена:

MB_OK (кнопка «OK»),
MB_OKCANCEL (кнопки «OK» 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 вводе из 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]

Имеет синонимы **Pause** и **PAUSE**. В качестве необязательного параметра можно использовать как непосредственный операнд-текст, так и адрес текста, например:

```
  pause «Нажмите любую клавишу…»
```

- **Макрокоманда порождения динамической переменной**

  new size; size≥0

---

[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 окна консоли и выполняемого файла .exe в проводнике и диспетчере задач Windows (см. пример Example_17.asm в masm 6.14.zip).

Макрокоманда порождает динамическую переменную размером `size` (это беззнаковый операнд формата `i32`) и возвращает адрес этой переменной в регистре `EAX`. Имеет синонимы **`New`** и **`NEW`**. При исчерпании кучи возвращается значение

```
nil equ 0
```

Например:

```
Node struc
Left   dd ?
Right  dd ?
Number dq ?
Node ends
.code
   new sizeof node
   mov [eax].Node.Left,nil
```

Размер выдаваемой макрокомандой **`new`** области 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`**. Имеет синоним **`heapblocksize`**

- **Макрокоманда уничтожения динамической переменной**

```
dispose address
```

Макрокоманда уничтожает динамическую переменную, адрес которой имеет формат `r32` или `m32`. Имеет синонимы **`Dispose`** и **`DISPOSE`**. Как и в Паскале, значение (ссылочной) переменной `address` при этом не меняется, например:

```
new sizeof node
dispose eax
mov eax,nil
```

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. К сожалению, данный механизм контроля утечки памяти работает правильно, только если все динамические переменные в программе одного размера.

- **Макрокоманда вывода даты**

```
OutDate[ln] [text]
```

Имеет синонимы **`outdate`** и **`OUTDATE`**. Эта макрокоманда выводит текущую дату в формате `dd.mm.yyyy`, если задан параметр-текст, то он выводится *перед* датой, например:

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

- **Макрокоманда вывода текущего времени**

```
OutTime[ln] [text]
```

Имеет синонимы **`outtime`** и **`OUTTIME`**. Эта макрокоманда выводит текущее время в формате `hh:mm:ss`, если задан параметр, то этот текст выводится *перед* временем, например:

```
.data
T  db "Сейчас ",0
.code
```

```
  SetTextAttr Yellow
  OutTimeln "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.), сделанными вызванной программой. Вызванная программа, однако, может открыть и своё собственное консольное окно, вызвав макрокоманду **NewConsole**, в этом случае после возврата новое окно закрывается, а старая программа продолжит выполнение в своём неизменённом окне. Вызванная программа может быть как консольным, так и графическим приложением Windows, например:

RunExe "C:\Program Files (x86)\winmine.exe"

К сожалению, перенаправление ввода/вывода символами > и < работать при этом не будет 🙁.

- **Макрофункция получения параметров командной строки**

  @GetCommandLine

Эта макрофункция возвращает (на регистре EAX) ссылку на командную строку запуска текущей программы, строка оканчивается нулевым символом, например:

```
C>masm 6.14\_Examples\MyProg.exe Par1 Par2
.code
  outstr @GetCommandLine()
Out: "masm 6.14\_Examples\MyProg.exe" Par1 Par2
```

- **Макрофункция генерации случайного числа**

  @Random([n])

Имеет синоним **@random**. В качестве *беззнакового* операнда n можно использовать форматы r32, m32 или i32. Эта макрофункция возвращает в регистре EAX (подставляемом на место своего вызова) псевдослучайное целое число из диапазона 0..n, если n опущено, то берётся n=Maxlongword. Например:

```
.code
  lea eax,[@Random(12)+1]
  OutWordLn eax,,"Случайный номер месяца = "
```

- **Макрокоманда установки генератора случайных чисел**

  Randomize

Имеет синоним **randomize**. Эта макрокоманда производит начальную установку генератора псевдослучайных целых чисел, используя системный таймер.

- **Макрокоманда вывода флагов**

  OutFlags

Эта макрокоманда без параметров предназначена для отладки программ на Ассемблере, она выводит текущие значения четырёх часто используемых флагов (сама она , естественно, не портит регистры и флаги):

```
.code
  OutFlags
```

Например, выводится `CF=0  OF=1  ZF=0  SF=1`

- **Макрокоманда выхода из программы**

  exit [<код возврата>]

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 *макрокоманду* **pause** для задержку окончания работы, 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** (для командного интерпретатора) уже стоит в конце этого пакетного файла. Код возврата такой же, как у оператора `halt` языка Free Pascal, если код опустить, предполагается ноль. Имеет синонимы **Exit** и **EXIT**.

Замечание. Все макрокоманды *вывода* (**outchar**, **outint**, **outstr**, **gotoxy**, **msgbox** и др.) не портят регистры и флаги. Макрокоманды *ввода* (**inchar**, **inint**, **readkey** и др.) не изменяют регистры (за исключением тех, на которых они возвращают результат), флаги меняет только макрокоманда **inint**, как было описано выше, она устанавливает определённые флаги в соответствие с результатом своей работы.

6.5.2. Структура программы на Ассемблере

Путь начинающего ассемблерщика не только долог, но ещё и тернист. Повсюду торчат острые шипы, дорогу преграждают разломы, ловушки и капканы. В тёмной чаще горят злые глаза, доносятся какие-то ухающие звуки и прочие неблагоприятные факторы, нагнетающие мрачную атмосферу и серьезно затрудняющую продвижение вперёд.

Крис Касперски ака мыщъх

Итак, программа Ассемблера представляет из себя набор предложений, являющихся командами, макрокомандами, директивами и предложениями резервирования памяти. По принципу фон Неймана в любом месте памяти может находится как команда, так и переменная, однако для удобства программирования лучше записывать команды и различные по смыслу данные в разные секции программы. Так, в секции с заголовком **.const** лучше хранить числовые и текстовые *константные* значения, по умолчанию эта секция при выполнении программы закрывается на запись, что позволяет предотвратить случайное изменение этих констант, например:

```
.const
T1 db "Ответ X="
N  dw 100; параметр задачи
```

Обратите внимание, что, например, константа Паскаля

```
const N=100;
```

ни в каких ячейках памяти *не храниться*, это просто *указание* компилятору о том, что всюду ниже по тексту программы надо имя `N` заменять на число 100. Аналогичное указание Ассемблеру записывается в виде директивы эквивалентности:

```
N equ 100
```

В секции с заголовком **.data** лучше хранить переменные, *возможно* имеющие начальные значения, как асякие переменные их значения могут далее меняться при счёте программы, например:

```
.data
x   dd  -1; переменная x может меняться в программе
    dw  2;  безымянная переменная с начальным значением
y   db  6 dup (0); массив из 6 изначально нулевых элементов
z   db  ?;      переменная z со случайным значением
```

На языке Free Pascal это такой раздел переменных:

```
var x: longint=-1;
{    dw 2; ??? Мистика }
     y: array[0..5] of byte=(0,0,0,0,0,0);
     z: char; {или byte, shortint, boolean}
```

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

```
.data? [1]
a    dd ?; переменная с неопределенным начальным значением
     db ?; безымянная переменная без начального значения
z    dw 2000000 dup (?); массив из 2000000 элементов
Bad  dw 1; ПРЕДУПРЕЖДЕНИЕ (warning), будет Bad dw ?
```

Выделение для неинициализированных переменных отдельной секции позволяет не хранить эту «пустую» секцию в загрузочном модуле, что позволяет уменьшить его размер в файловой системе («на диске»).[2] 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
a  dw  -1
   mov eax,1; безымянная область памяти со значением команды
b  db  -200; -200=256-200=56
c  dd  a; c=адрес(a),[3] можно и c dd offset a
```

Исключение составляют команды перехода (кроме команд *косвенного* перехода) и команды с метками (см. разд. 6.7), их в секциях данных указывать нельзя. Обычно задание команд в секции данных не имеет большого смысла и используется редко. Отметим, что по умолчанию секции констант и данных закрыты на выполнение, поэтому использовать константы и переменные в качестве команд просто так не получится.

Языки высокого уровня поступают так же, например, язык Free Pascal для описания

```
const a: byte=1;
var x,y: word; b: int64=-13;
```

разместит переменную a и переменную b в секции **.data**, а переменные x и y в секции **.data?**

И, наконец, в секции с заголовком **.code** [4] записываются команды (и макрокоманды) программы, например:

```
.code
Start: ❶
   outstrln "Начало программы"
   mov eax,1
   outint eax
```

---

[1] В операционных системах семейства Unix эта секция называется .bss

[2] При размещении этой секции в памяти перед началом счёта она обычно всё-таки заполняется нулями. Это делается не для удобства программиста, а из соображений безопасности, так как в этой памяти могли сохраниться (секретные) данные от предыдущей прораммы (такие прецеденты известны).

[3] Сам компилятор с Ассемблера формирует только **offset**, это расстояние до переменной от начала секции. "Настоящий" адрес получает только редактор внешних связей (или даже загрузчик), прибавляя к смещению адрес начала секции в памяти ЭВМ (см Главу 9).

[4] В операционных системах семейства Unix эта секция называется .text (это текст программы).

Обычно начало программы на Ассемблере в 32-битном режиме помечается меткой `Start` ❶, подробно об этом будем говорить далее. По умолчанию эта секция при счёте закрывается на запись, что позволяет предотвратить случайное изменение команд.

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 вместе с командами часто хранят и различные *константы*. Так поступают многие компиляторы с языков высокого уровня, защищая константы от изменения. Заметим, что у программиста на Ассемблере, в соответствие с принципами фон Неймана, имеется возможность открыть секции команд и констант на запись, а секции данных на выполнение. Как это сделать, описано в главе, посвящённой модульному программированию.

В секции кода не запрещается и резервировать память под 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` (хотя эту команду-префикс в нужных случаях ставит сам компилятор с Ассемблера):

```
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;  Ассемблер вставит префикс 66h mov eax,1
```

Начало описания новой секции отмечает конец предыдущей, последняя секция закрывается директивой конца модуля **end**. Допускается задавать в модуле на Ассемблере несколько одинаковых секций, при этом каждая следующая секция Ассемблером объединяется (добавляется в конец, «подклеивается») к предыдущей такой же секции. Секции могут располагаться в любом порядке, компилятор склеит одноимённые секции и запишет их в нужном порядке (в объектный модуль). Программист обычно не знает, как будут располагаться секции в памяти при счёте программы,[1] однако известно, что каждая секция начинается с новой страницы (адрес её начала кратен 4096).

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:[2]

```
_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
```

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

```
A dd ?
B dw -8; имеет начальное значение
X db ?
```

На языке Free Pascal нужная программа выглядит, например, так:

---

[1] В каждой системе программирование есть некоторое стандартное размещение секций в памяти ЭВМ, но программист может и изменить его.

[2] Когда-то давно секции на самом деле были сегментами, оттуда и осталось это название, сейчас это просто участки памяти.

```
program P(input,output);
var A: Longint; B: Smallint=-8; X: Shortint;
begin {$I+}{R-} { Режимы работы без контроля }
  Write('Введите целое A='); Read(A);
  X := (2*A - sqr(A+B)) mod 7;
  Writeln('Ответ X=',X:8)
end.
```

В переменной X будет получаться результат работы. Это *короткое* знаковое 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 секции программы (подробные пояснения будут даны сразу же вслед за текстом этого примера):

```
include console.inc
comment *
директива include  вставляет в начало программы
содержимое файла с именем console.inc,
содержащего директивы и макроопределения
*
.data?
A   dd   ?; В УМ-3 это: пусть A в ячейке 100
B   dw  -8; В УМ-3 это константа за СТОП
X   db   ?; В УМ-3 это: пусть X в ячейке 101
.code
; X := (2*A - (A+B)^2) mod 7
Start:
; Write('Введите целое A='); Read(A)
  inint  A,'Введите целое A='
; Пусть введено «хорошее» A, т.е. CF=0
  mov    ebx,A;    ebx := A
  movsx  eax,B;    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, т.е. CF=OF=0
  add    ebx,ebx;  ebx := A+A = 2*A
; Пусть 2*A помещается в ebx, т.е. OF=0
  sub    ebx,eax;  ebx := 2*A - (A+B)^2
; Пусть 2*A-(A+B)^2 помещается в ebx, т.е. OF=0
  mov    eax,ebx;  делимое
  cdq;             <edx:eax> := int64(eax)
  mov    ebx,7;    32-битный делитель
  idiv   ebx;      edx := (2*A - (A+B)^2) mod 7
  mov    X,dl;     ответ длиной в байт
; Writeln('Ответ X=',X:8)
  outintln X,8,"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. Следующее предложение (это макрокоманда **inint**) выводит приглашение для ввода переменной A и вводит значение этой переменной.

Далее идёт непосредственное вычисление правой части оператора присваивания. Задача усложняется тем, что величины A и B имеют разную длину (4 и 2 байта соответственно) и непосредственно складывать их нельзя. Приходится командой

```
movsx eax,B; eax := Longint(B)
```

преобразовать длинное (**dw**) *знаковое* целое число B в сверхдлинное (**dd**) целое на регистре EAX. Далее вычисляется значение выражения $(A+B)^2$. Произведение занимает два регистра (регистровую пару) <EDX:EAX>. Пока для простоты будем предполагать, что в старшей части произведения (регистре EDX) находится *незначащая* часть произведения и в качестве результата произведения можно взять только значение регистра EAX (напомним, что работа ведётся в режиме {R-} без контроля выхода значений за допустимые диапазоны). Далее вычисляется делимое $2*A-(A+B)^2$, оно пересылается на регистр EAX, и можно приступать 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 <DX:AX> **mod** 7 *всегда* поместиться в отведённое ему место (регистр DX), так как он по абсолютной величине меньше 7. В то же время частное <DX:AX> **div** 7 может не поместиться в регистр AX, 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 EDX можно присвоить переменной X, а так как X имеет длину всего один байт, то берётся только значение регистра DL – самой младшей части регистра EDX.

Последним предложением в секции кода является макрокоманда

```
exit
```

Эта макрокоманда заканчивает выполнение нашей программы, она эквивалентна выходу программы на Паскале на конечный **end.** Это **exit** 0, т.е. с хорошим кодом возврата.

Вообще говоря, при запуске на счёт головная программа на Ассемблере (см. разд. 9.1) вызывается как (головная) функция (см. разд. 6.10) и возможен возврат из программы не по макрокоманде **exit**, а по машинной команде **ret**. В этом случае для выдачи кода возврата надо предварительно выполнить команду **mov** eax,<код возврата>.

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, как на Паскале. 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 <EDX:EAX>, а в написанной программе результат произведения брался только из регистра EAX, т.е. предполагалось, что на регистре EDX находятся только *незначащие* биты произведения. По-хорошему надо было бы после команды умножения проверить условие, что флаги OF=CF=0, это гарантирует, что в EDX содержится *незначащая* часть произведения. И, наконец, не проверялось, что макрокоманда **inint** возвратила правильное число (установила CF=0). В *учебных* программах такие проверки делаются не все-

гда, но в «настоящих» программах, которые Вы будете создавать на компьютерах, эти проверки являются обязательными.

Теперь, после изучения арифметических операций, перейдём к рассмотрению команд *переходов*, которые понадобятся нам для программирования условных операторов и циклов. Напомним, что после освоения материала этой книги Вы должны уметь отображать на Ассемблер большинство конструкций языков программирования высокого уровня (хотя бы Паскаля 🙂).

6.6. Переходы

Выходов нет, есть только переходы.
Григорий Ландау

Перед переходом сначала посмотрите налево, потом направо (если Вы не в Англии 🙂).

В компьютерах по принципу фон Неймана реализовано *последовательное выполнение команд*. В соответствии с ним, перед выполнением текущей команды счётчик адреса устанавливается на следующую (ближайшую с бóльшим адресом) команду в оперативной памяти.

Таким образом, команды программы, расположенные последовательно в оперативной памяти, обычно выполняются друг за другом. Однако, как и в учебной машине УМ-3, такой порядок выполнения программы может изменяться с помощью *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 очень быстро по одному разу выполнить все команды в оперативной памяти. Исходя из этого, большинство алгоритмов содержат внутри себя циклы и условные операторы,[1] 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). Эти переходы будут изучаться в главе 7.

Начнём рассмотрение *команд переходов* для компьютеров изучаемой нами архитектуры. Напомним, что логический адрес начала *следующей* выполняемой команды хранится в регистре `EIP`. Следовательно, для осуществления любого перехода нужно занести в этот регистр *новое* значение. По способу изменения значения регистра `EIP` различают *относительные* и *абсолютные* переходы. При относительном переходе (relative jump) происходит *знаковое* сложение содержимого регистра `EIP` с некоторой величиной, например,

```
EIP := (EIP ± Value) mod 2^32
```

Здесь полезно вспомнить доступ к памяти с помощью *базового регистра*, в качестве которого здесь и выступает регистр `EIP`. При абсолютном переходе (absolute jump) 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 (short) переходе величина этой *знаковой* константы не превышает по размеру одного байта (напомним, что она обозначается `i8`, и лежит в диапазоне от `–128` до `+127`):

```
EIP := (EIP ± i8) mod 2^32,
```

[1] Существует целый класс так называемых функциональных языков программирования, в которых циклов и условных операторов нет.

а при длинном (long) переходе эта *знаковая* константа имеет размер `i32` (двойное слово): [1]

`EIP := (EIP ± i32) mod` 2^{32}

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

Следующей основой для классификации абсолютных переходов будет месторасположение величины `Value`, которая присваивается регистру `EIP`. При прямом переходе (direct jump) эта величина является просто числом-адресом команды (здесь это *непосредственный* адрес в самой команде `i8` или `i32`). При косвенном переходе (indirect jump) 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];` это `EIP := m32↑`

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

Заметим, что в языках высокого уровня есть косвенная адресация переменных (через ссылки), но нет косвенных переходов **`goto`**, так как это выглядело бы совсем странно, например

```
var y: Longword; x: Pointer;
...x:=@y; x↑:=y+6; {Это понятно}
...goto x↑; {???}
{Это EIP:=x↑, куда переход ???}
```

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, относительные переходы бывают только прямыми, кроме того, абсолютные переходы бывают только длинными.

6.7. Команды переходов

А «язык» процессоров x86, между прочим, очень интересен. На сегодняшний день они имеют едва ли не самую сложную систему команд, дающую системным программистам безграничные возможности для самовыражения. Прикладные программисты даже не догадываются, сколько красок мира у них украли компиляторы!

Крис Касперски ака мыщъх

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

6.7.1. Команды безусловного перехода

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

Эдгар Дейкстра.
«О вреде оператора Go To»

---

[1] Поставив перед командой перехода префикс `67h`, можно использовать и константу длиной в два байта, но большого смысла это не имеет. Экономия по памяти составит только один байт, но на выполнения префикса процессор потратит лишний такт своей работы 😕.

Рассмотрим сначала команды *безусловного* перехода (unconditional transfer), которые всегда передают управление в указанную в них точку программы. На языке Ассемблера все команды такого перехода записываются в виде

```
jmp op1
```

Здесь операнд `op1` имеет форматы, показанные в таблице 6.1.[ii] [см. сноску в конце главы]

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

Обратите внимание, что в языке машины не реализована команда абсолютного прямого перехода. Здесь дело в том, что на современных ЭВМ программа обычно пишется так, чтобы она могла выполняться на *любом свободном месте* оперативной памяти. Таким образом, программист (и компилятор с языка высокого уровня) не знает, в каком конкретно месте памяти будет находится та или иная команда, более подробно об этом говорится в разд. 9.3.

Рассмотрим теперь, как на языке Ассемблера задаётся 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 с меткой L
```

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

```
L:  add rbx,rbx;
     . . .
    . . .        i8 или i32
                 (со знаком!)
    . . .
$→ jmp L; L=i8 или i32
EIP→
```

Здесь символом `$` (так принято в Ассемблере) обозначено текущее значение так называемого счётчика размещения (location counter), это адрес первого байта *текущей* команды в секции кода или адрес первого байта *текущей* переменной в секциях данных. Таким образом, для секции кода во время счёта `$=EIP-<длина текущей команды>`, поэтому выполнение команды `jmp $` есть лучший способ зациклиться 🙂. Как показано, в момент выполнения команды перехода счётчик адреса `EIP` уже указывает на *следующую* команду, а позиция текущей команды `$` – на команду перехода, это существенно при вычислении величины смещения в команде относительного перехода. К счастью, всю эту работу по выбору нужного формата команды перехода и вычисления необходимого смещения выполняет компилятор Ассемблера при трансляции программы. Для программиста эта особенность будет существенной далее, при изучении команд вызова процедуры и возврата из процедуры. [iii] [см. сноску в конце главы]

Заметим, что программист и сам может вычислить нужное смещение (`i8` или `i32`) и явно задать его в команде перехода в виде:

```
        ...
        ...
        ...
; ↓ EIP := EIP-9-<длина команды jmp $-9>
    jmp $-9
EIP →
```

Текущая позиция `$` в секции данных может использоваться и для определение длины области памяти, например:

```
.data
X dw 1,2,3,4 dup (0); X 14 байт
Y dd ? ;              Y 4 байта
Z equ $-X; Z=18 байт между Z и X
```

Учтите, что языки высокого уровня (Free Pascal, C и т.д.), размещая в секции данных описанные программистом переменные, обычно располагают каждую переменную с адреса памяти, кратного 8 или 16. Разумеется, это не касается элементов массивов, они идут в памяти друг за другом подряд. Причину этого легко понять, если учесть, что за одно обращение к памяти процессор получает не один, а 8 или бóльшее количество байт, поэтому, когда переменная пересекает границу 8 байт, то надо сделать *два* обращение к памяти.

Для аналогичного выравнивания переменных на языке Ассемблера программистом может использоваться директива **`align`** `{1,2,4,8,16}`. Директива увеличивают счётчик размещения, чтобы он стал кратным параметру, т.е. выровнен в памяти на границу соответствующего числа байт (при этом пропущенные байты в секциях **`.data`** и **`.code`** заполняются нулями), например:

```
.data
X dw 1,2,3,4 dup (0)
  align 16
Y dd ?
Z equ $-X; Z=24 байта между Z и X
```

Для директивы **`align`** `2` есть краткий синоним **`even`**. При использовании директивы **`align`** размер программы увеличивается, но скорость работы возрастает.

При близком переходе формат для операнда (`i8` или `i32`) выбирается компилятором Ассемблера автоматически, в зависимости от расстояния в байтах между командой перехода и командой с указанной меткой. Формат `i8` даёт более короткую команду, так что лучше далеко не прыгать 🙂.

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 эти значения 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, по существу, только одна такая команда. На этом закончим рассмотрение команд безусловного перехода.

6.7.2. Команды условного перехода

Когда необходимо сделать выбор, а вы его не делаете, – это тоже выбор.
Уильям Джеймс
«Принципы психологии»,1890

Все команды условного перехода (conditional jump) выполняются по схеме, которую на Паскале можно записать как

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

и производят *относительный прямой* переход, если условие перехода выполнено, 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), например:

```
if x<y then goto L
```

Если обозначить `rez=op1-op2`, то (когда значение `rez` существует)[1] этот оператор условного перехода 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`.

Команда условного перехода в языке Ассемблера имеет такой вид:

```
j<мнемоника перехода> op1
```

Здесь `op1` может иметь форматы `i8` или `i32`,[2] причём `i8` *знаково* расширяется до `i32`. Команда реализует *относительный прямой* переход и выполняется по схеме

```
EIP := (EIP + Longint(op1)) mod 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 (или 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 `x=0FFh` и `y=01h`. Как *знаковые* числа `x=-1` меньше `y=1`, а как *беззнаковые* числа `x=255` больше `y=1`.

Заметим, что в языках высокого уровня нужное (знаковое или беззнаковое) сравнение берётся компилятором, исходя из типов данных:

```
var x: byte=255; y: shortint=-1;
if x>1 {true}  then ...;
if y>1 {false} then ...;
```

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 `op1>op2`» 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.

Как и для команд относительного безусловного перехода, тип перехода (`i8` или `i32`), выбирается компилятором Ассемблера автоматически, в зависимости от расстояния от счётчика адреса `EIP` до

---

[1] В дискретной математике в общем случае `x<y ≠ x-y<0`.

[2] Некоторые команды условных переходов (**`loop`**, **`jecxz`** и другие), к сожалению, допускают только операнд формата `i8`, это будет подробно описано далее.

нужной метки (напомним, что при выполнении команды относительного перехода EIP указывает на начало *следующей* команды). Отметим, что несмотря на многочисленность команд условного перехода, для кодирования условия перехода в коде операции используется всего четыре бита (если не принимать в расчёт команду условного перехода по регистру ECX).

Таблица 6.2. Мнемоника команд условного перехода

КОП	Условие перехода	
	Логическое условие перехода	Результат (rez) команды вычитания или соотношение операндов op1 и op2 команды сравнения
je **jz**	ZF = 1	rez = 0 или op1 = op2 Результат = 0, операнды равны
jne **jnz**	ZF = 0	rez <> 0 или op1 <> op2 Результат <> 0, операнды не равны
jg **jnle**	(SF=OF) **and** (ZF=0)	rez > 0 или op1 > op2 Знаковый результат > 0, op1 **больше** op2
jge **jnl**	SF = OF	rez >= 0 или op1 >= op2 Знаковый результат >= 0, т.е. op1 **больше или равен** (**не меньше**) op2
jl **jnge**	SF <> OF [1]	rez < 0 или op1 < op2 Знаковый результат < 0, т.е. op1 **меньше** (**не больше или равен**) op2
jle **jng**	(SF<>OF) **or** (ZF=1)	rez <= 0 или op1 <= op2 Знаковый результат <= 0, т.е. op1 **меньше или равен** (**не больше**) op2
ja **jnbe**	(CF=0) **and** (ZF=0)	rez > 0 или op1 > op2 Беззнаковый результат > 0, т.е. op1 **выше** (**не ниже или равен**) op2
jae **jnb** **jnc**	CF = 0	Беззнаковые операнды op1 >= op2 op1 **выше или равен** (**не ниже**) op2
jb **jnae** **jc**	CF = 1	Беззнаковые операнды op1 < op2 op1 **ниже** (**не выше или равен**) op2
jbe **jna**	(CF=1) **or** (ZF=1)	Беззнаковые операнды op1 <= op2 op1 **ниже или равен** (**не выше**) op2
js	SF = 1	Знаковый бит разультата (7-й, 15-й или 31-й) равен единице
jns	SF = 0	Знаковый бит разультата (7-й, 15-й или 31-й) равен нулю
jo	OF = 1	Флаг переполнения равен единице
jno	OF = 0	Флаг переполнения равен нулю
jp **jpe**	PF = 1	Флаг чётности равен единице
jnp **jpo**	PF = 0	Флаг чётности равен нулю [iv] [см. сноску в конце главы]
jcxz **jecxz**	CX = 0 ECX = 0	Значение регистра CX/ECX равно нулю

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

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

[1] Необходимо учесть, что, вообще говоря, условие `op1<op2` совсем не означает, что существует `rez=op1-op2` и этот `rez<0`. Как показано в Таблице 6.2 для `op1<op2` надо `SF<>OF`, а чтобы *существовал* `rez<0` надо более сильное условие `(OF=0) and (SF=1)`.

На языке Free Pascal решение этой задачи можно записать в виде программы

```
const N=10000
var A,X: Longint;
begin {$I-} { Ручной» контроль ввода }
  Read(A);
  if IOResult<>0 then begin
    Writeln('Плохое значение A!');
    halt
  end;
{$R+} { Контроль выхода за допустимый диапазон }
{L:}
  if A>N then ❶ X:=(A+1)*(A-1) else
{L1:}
  if A=N then ❷ X:=4
{L2:}    else ❸ X:=(A+1) mod 7;
{Pech:}
  Writeln(X)
end.
```

Здесь в виде комментариев указаны метки, которые будут использоваться в программе на Ассемблере. 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 `{$R-}` выдастся неверный ответ, а в режиме `{$R+}` будет аварийное завершение программы. В программе на Ассемблере мы будем всегда выдавать или правильный ответ, или диагностику об ошибке (вообще говоря, с возможностью продолжить выполнение программы):

```
include console.inc
N  equ 10000
.data; можно и .data?
A  dd ?; var A: Longint
X  dd ?; var X: Longint
.code
Start:
   inint A; Read(A)
   jnc   L; Введено хорошее A
Error:
   outstrln 'Плохое значение A!'
   exit 1; Плохой конец программы
L: mov  eax,A; eax:=A
   mov  ebx,eax; ebx:=A
   inc  eax; eax:=A+1
   jo   Error; A+1 вне допустимого диапазона
   cmp  ebx,N; Сравнение A и N
   jle  L1; Вниз по первой ветви then ❶
   dec  ebx; ebx:=A-1, всегда хорошо, т.к. N=10000
   imul ebx; <edx:eax>:=(A+1)*(A-1)
   jo   Error; (A+1)*(A-1) не помещается в eax
   mov  X,eax; Результат берётся только из eax
 Pech:
   outintln X; Вывод результата
   exit 0; Хороший конец программы
L1:jl   L2; Вниз по второй ветви then ❷ вычисления X:=4
   mov  X,4
   jmp  Pech; На вывод результата
L2: mov  ebx,7; Третья ветвь else ❸ вычисления X, A+1 на eax
   cdq; <edx:eax>:=int64(A+1)
   idiv ebx; edx:=<edx:eax> mod 7
```

```
;              eax:=<edx:eax> div 7
    mov  X,edx; X:=(A+1) mod 7
    jmp  Pech; На вывод результата
    end Start
```

В этой программе сначала кодировалось вычисление ❶ по первой ветви алгоритма, затем ❷ по второй и, наконец, ❸ по третьей. Программист, однако, может выбрать и другую последовательность кодирования ветвей, обычно это не влияет на суть дела.[1] Далее, предусматривается выдача аварийной диагностики (для простоты все диагностики одинаковые), 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. В нашем примере, однако, числа *знаковые*, поэтому всё в порядке.

В языке Ассемблера MASM, начиная с версий 6.xx, введена директива **`.if`** для облегчения программирования условных операторов из языков программирования высокого уровня:[2]

```
.if <логическое выражение>[3]
<предложения Ассемблера>
[.elseif <логическое выражение>
<предложения Ассемблера> ]
[.else
<предложения Ассемблера> ]
.endif
```

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

```
X   dd ?
   . . .
.if !CARRY? && eax<X
    add eax,10
.else
    add ebx,eax
.endif
```

будет преобразована компилятором Ассемблера в такой набор предложений:

```
jc  @C0001; CF=1 => на ветвь else
cmp eax,X;  здесь CF=0
jae @C0001; на ветвь else
add eax,10; здесь eax<X
jmp @C0002
```

---

[1] Вообще говоря, лучше первой помещать *наиболее вероятную* ветвь, тогда ЭВМ работает более эффективно, о чём будет говориться в другой главе.

[2] Директивы, имена которых начинаются с точки, реализованы в виде так называемых стандартных макрокоманд, что это такое, будет изучаться в главе, посвящённой макросредствам Ассемблера.

[3] Не запутайтесь, в нашем Ассемблере есть и условный макрооператор **`if else endif`** (имена без точек), он описан в Главе 11.

```
@C0001:  ;  ветвь else
add ebx,eax
@C0002:
```

на Паскале он эквивалентен условному оператору

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

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 переменных, описанных при помощи *знаковых* директив резервирования памяти **sbyte**, **sword** или **sdword**, а иначе он будет считать значения *знаковыми*, например:

```
X dword ?; или X dd ?
     . . .
.if X<1;  тогда cmp X,1
jae @C0001
; ----------------------
X sdword ?; «знаковое» dd
     . . .
.if X<1;  тогда cmp X,1
jge @C0001
```

По похожим правилам различаются знаковые и беззнаковые величины и при вычислении логических выражений в условных макрооператорах, о чем будет рассказано в главе, посвященной макросредствам языка Ассемблер. Помните, однако, что в *машинных* командах (**add**, **cmp** и т.д.) переменные, описанные, например, при помощи директив **dd**, **dword** и **sdword** полностью эквивалентны.

Начиная с процессора Intel 486 появилась «хитрая» команда *условной пересылки* **cmpxchg** (**CoMP**are and e**XCH**an**G**e), 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

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 (если неявный операнд не равен первому операнду) `<неявный операнд>:=op1`. Эта команда, как и, например, **xchg**, является неделимой операцией, т.е. при её выполнении не снимается блокировка с шины обмена данными с оперативной памятью.

Начиная с процессора Intel P5 появилась команда *условного обмена* **cmpxchg8b** (**CoMP**are and e**XCH**an**G**e **8 B**yte – 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 около 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. На новых процессорах, естественно, появилась и команда **cmpxchg16b**.

Начиная с процессора Pentium Pro (1995 год) можно использовать новые команды *условной пересылки* **CMOVcc** (**C**ondition **MOV**):[1]

```
CMOVcc op1,op2; if cc then op1:=op2
```

где после **CMOV** указывается мнемоника любой команды условного перехода, например, **cmovg**, **cmovae**, **cmovz** и т.д. Допустимые форматы операндов (обратите внимание, что нет операндов форматов i16 и i32):

op1	op2
r16	r16, m16
r32	r32, m32

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 показанный условный оператор Паскаля можно переписать без использование 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 байта

```
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 одном байте «на будущее» выработанное условие (в частности, значение флага) для последующих команд перехода.[v] [см. сноску в конце главы]

В архитектуре Intel существует только команда условной *пересылки* **CMOVxx**, а, например, в процессорах ARM, используемых в планшетах и смартфонах, есть возможность условного выполнения почти любой команды. Например, просто представьте, что в нашем Ассемблере есть такие «условные» команды **CADDxx**, **CSUBxx**, **CMULxx**, **CCALLxx** и т.д. Заметим, однако, что в архитектуре Intel при работе на *векторных регистрах* существует аналогичный механизм регистров масок (см. разд. 17.6), они позволяют выполнять условно почти все операции, причём условие задаётся для каждого элемента на векторном регистре независимо.

6.7.3. Команды цикла

Всё в мире повторяется, и возвращается ветер на круги своя.

Соломон. «Экклезиаст»

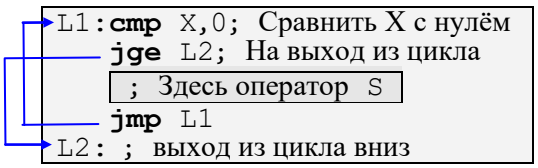
1. Начальник всегда прав.
2. Если начальник не прав, см. Пункт 1.

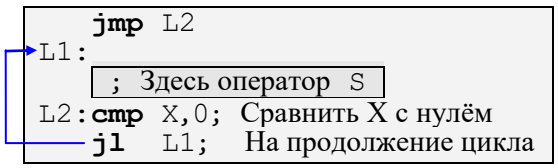
Цикл с постусловием 

Для организации циклов на Ассемблере используются команды переходов. Например, цикл языка Паскаль с предусловием `while X<0 do S` для целой *знаковой* переменной X можно реализовать в виде следующих фрагментов на Ассемблере:

---

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





В режиме без оптимизации компиляторы реализуют этот цикл по первой схеме, а с оптимизацией – по второй. В первом случае в внутри цикла выполняются две команды перехода, а во втором – только одна. Заметьте, что вторая реализация является «замаскированным» циклом *с постусловием*, что может увеличить скорость работы программы.

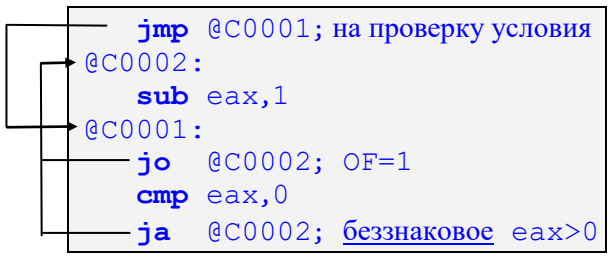
В языке Ассемблера, начиная с версий 6.xx, введена директива **.while** для облегчения программирования цикла с предусловием из широко распространённых языков программирования высокого уровня, синтаксис этой директивы:

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

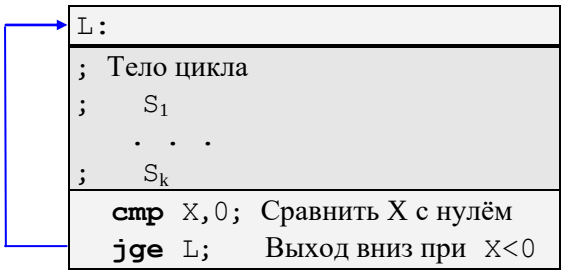
Например, конструкция

```
.while OVERFLOW? || eax>0; Это (OF=1) or (eax>0)
   sub eax,1
.endw
```

будет преобразована компилятором Ассемблера в такую последовательность команд:



Оператор цикла с постусловием **repeat** S_1; S_2; ... S_k **until** X<0 для целой *знаковой* переменной X можно реализовать в виде такого фрагмента на Ассемблере:



В языке Ассемблера, начиная с версий 6.xx, введена директива **.repeat** для облегчения программирования цикла с постусловием из широко распространённых языков программирования высокого уровня, синтаксис этой директивы:

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

Например, конструкция

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

будет преобразована компилятором Ассемблера в такую последовательность команд:

```
@C0001:
    sub  eax,1
    test eax,0; лучше or eax,eax
; команда test лишняя, но компилятор тупой 🙂
    je  @C0002; if eax=0 then goto @C0002
    jc  @C0001; if CF=1  then goto @C0001
@C0002:
```

Внутри циклов **.while … .endw** и **.repeat … until** можно также использовать условную или безусловную директиву досрочного выхода из цикла

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

и директиву

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

Эти директивы производят условный или безусловный переход на проверку логического условия продолжения выполнения циклов. Директивы **.if**, **.while**, **.repeat**, **.break** и **.continue** в программах этой книге использоваться не будут.

Все эти стандартные макрокоманды относятся к так называемому *высокоуровневому ассемблированию* (HLA – High Level Assembler). Существуют даже специальные высокоуровневые Ассемблеры, например, Ассемблер HLASM (High-Level Assembler) фирмы IBM. Следует также отметить, что многие языки высокого уровня позволяют использовать так называемые встраиваемые функции (intrinsic functions), которые, по существу, являются мнемоническими обозначениями команд Ассемблера (в основном для работы с векторными регистрами процессора).

Итак, цикл с постусловием требует для своей реализации на одну команду (и на одну метку) меньше, чем цикл с предусловием, а значит его использование предпочтительнее.

Например, компиляторы с языка C, к удивлению адептов этого языка, реализуют их любимые циклы **while** и **for** в виде «чистых» циклов **repeat**,[1] если уверены, что тело цикла должно выполниться хотя бы один раз, в противном случае, как было показано выше, *имитируют* цикл **repeat**, первой же командой передавая управление на проверку выхода из цикла. Кроме того надо иметь в виду, что процессоры фирмы Intel оптимизированы под эффективное выполнение именно циклов с постусловием (**repeat** и **loop**), т.е. с передачей *в конце цикла* управления *назад* по тексту программы.

В том случае, если число повторений тела цикла известно до начала исполнения этого цикла, то в языках высокого уровня (например, в Паскале) наиболее целесообразно применять *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
```

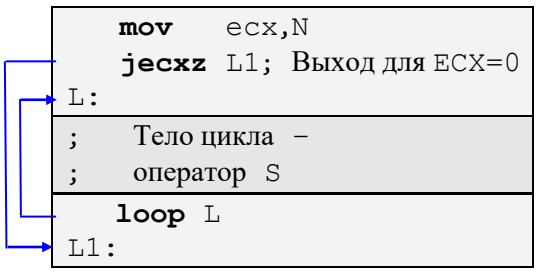
Как видим, регистр ECX (который так и называется (расширенным) регистром-счётчиком цикла – Extended loop Counter), используется этой командой именно как параметр цикла. Лучше всего эта команда цикла подходит для реализации цикла с параметром языка Паскаль вида

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

Этот оператор можно реализовать таким фрагментом на Ассемблере:

---

[1] В языке C аналогом цикла **repeat** является цикл **do**{…}**while**(<лог.выр.>), который, в отличие от Паскалевского **repeat**, выходит из цикла, когда логическое выражение принимает значение **false**.



```
    mov   ecx,N
    jecxz L1; Выход для ECX=0
L:
;    Тело цикла –
;    оператор S
    loop L
L1:
```

Обратите внимание: так как цикл с параметром языка Паскаль является циклом с предусловием (может не выполняться ни разу), то сначала у нас 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 только короткий переход, поэтому, если расстояние до метки L1 велико, то следует использовать пару команд `test ecx,ecx` и `jz L1`, но флаги при этом *меняются*. 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 20-30 машинных команд).

Описанная выше команда цикла выполняет его тело 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).

На современных процессорах Intel команда `loop L` проигрывает «прямому» программированию, она выполняется *медленнее*, чем пара команд `sub ecx,1` и `jnz L`. Дело в том, что по так называемой технологии макрослияния (macro-fusion) некоторые пары команд могут «спариваться». Такие пары команд сливаются в одну микрооперацию (в один моп) и выполняются параллельно на двух обрабатывающих устройствах (портах) конвейера за один такт работы процессора. А вот команда `loop L` разлагается на 5-6 мопов, и, кроме того, реализует только *короткий* (i8) условный переход, в то время как команда `jnz L` может выполнять как короткий, так и длинный (i32) переход. Видно, что фирме Intel можно не стараться сделать команду **loop** быстрее, поэтому сейчас она практически не используется компиляторами с языков высокого уровня. Единственным преимуществом команды **loop** является то, что она не портит флаги и немного короче.

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 бóльшую 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 (spatial locality) команд и данных. Для команд может существовать и временна́я локальность (temporal locality), 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<=500000`, 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=500000; {$R+} {$I-} {$goto+}
  label L,L0,L1,L2,Pech;
  var   N: longword; eax,ecx: longint; S: longint=0;
begin
  Write('Введите N<=500000 : ') ;
  Read(N);
```

```
  if IOResult<>0 then begin
L:   Writeln('Введено плохое число!'); halt
  end;
L0:
  if N > MaxN then begin
    Writeln('Ошибка – большое N!'); halt
  end;
  if N > 0 then begin
    Writeln('Вводите целые числа:');
L1: for ecx:=N downto 1 do begin
L2:   Read(eax);
      if IOResult<>0 then goto L;
      if (-2000<=eax) and (eax<=5000) then S:=S+eax
    end
  end;
Pech: Writeln('S=',S)
end.
```

На Ассемблере можно предложить следующее решение этой задачи.

```
include console.inc
MaxN equ 500000; Аналог const MaxN=500;
.data
   N dd  ?; var N: longword;
   S dd  0; var S: longint=0;
.code
Start:
   ininтln N,"Введите N<=500 : "
   jnc   L0
L: outstrln "Введено плохое число!"
   exit  1
L0:cmp   N,MaxN
   jbe   L1
   outstrln "Ошибка – большое N!"
   exit  1
L1: mov   ecx,N; Счётчик цикла
   cmp   ecx,0
   je    Pech;  На печать при N=0
   outstrln 'Вводите целые числа:'
L2:inint eax;    Ввод очередного числа
   jc    L;     Плохое число
   cmp   eax,-2000
   jl    L3
   cmp   eax,5000
   jg    L3;    Проверка eax in [-2000..5000]
   add   S,eax; Суммирование
;  Переполнения не бывает: S<500*5000<Maxlongint
;  jo L - не надо
L3:dec   ecx
   jnz   L2;    Цикл
Pech:
   outintln S,,'S='
   exit
   end Start
```

Как видно, так как в программе на Паскале заданы директивы {$R+} и {$I-} то и приведённая программа на Ассемблере также проверяет корректность ввода и выход результатов операций за допустимые диапазоны.

Выход *частичной* суммы за границу допустимого диапазона (с установкой флага `OF=1`) для знаковых чисел в дополнительном коде совсем не означает, что и *итоговая* сумма всех элементов массива будет неправильной! Действительно, пусть, скажем частичная сумма целых чисел превысила `MaxInt` и стала неправильной. Дальше, однако, вполне вероятно, что будут суммироваться *отрицательные* числа и сумма снова войдёт в допустимый диапазон. Вообще говоря, частичная сумма может, например, пять раз выйти за верхнюю допустимую границе `MaxInt`, а затем пять раз вернуться назад, и конечный ответ станет правильным. Можно реализовать алгоритм суммирования, который подсчитывает число таких «выходов и возвратов», определяя, получится ли итоговая сумма правильной. Ясно, что для суммирования *беззнаковых* чисел это не работает, так как они не могут быть отрицательными, а вот когда в алгоритме есть и сложение и вычитание беззнаковых чисел, то при выходе за допустимый диапазон они ведут себя так же, как и знаковые.

Максимальное количество вводимых чисел задано в программе директивой эквивалентности `MaxN equ 500000`, 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 именно *макроконстантой* станет ясно при описании макросредств). Таким образом, это почти полный аналог описания константы в языке Паскаль (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).

Макроконстанты эквивалентны целочисленным константам языков высокого уровня формата **dd** (4 байта), поэтому их, естественно, нельзя переопределять (менять). Однако, кроме числовых макроконстант директива **equ** может задавать и текстовые макропеременные, в этом случае параметр директивы **equ** *рассматривается* как текстовая строка, которую можно не заключать в кавычки и апострофы. Строка должна быть сбалансированная по скобкам и кавычкам. В параметре директивы **equ** можно использовать имена, описанные как до, так и после этой директивы. Макропеременные, естественно, можно далее в программе менять. Тип определяется по первой директиве с данным именем, при этом смотрится тип выражения в правой части, например:

Числовая макроконстанта	
`A equ 10;`	`const A=10;`
`A equ 20;`	Ошибка
`B equ 1abc;`	Ошибка
`C equ 1+2;`	`"const" C=1+2=3;`
`D equ 1,2;`	Ошибка
`E equ 5000000000;`	Ошибка >4 байт

Текстовая макропеременная	
`N equ abc;`	`const N: String='abc';`
`N equ def;`	`N:='def'`
`N equ 100;`	`N:='100'`
`N equ 200;`	`N:='200'`
`M equ a+1;`	`const M: String='a+1';`
`L equ '1,2';`	`const L: String=''1,2''`

Кроме того, в Ассемблере есть ещё и директиве **textequ** (операнд которой всегда заключается в угловые скобки), предоставляющая большие возможности, здесь всё весьма запутано ☹. Иногда перед подстановкой параметра на место макроконстанты и макропеременной производится также некоторые *вычисления* операнда директивы эквивалентности, подробнее об этом мы будем говорить в главе 11.

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 формате **dd** 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. Для простоты правильность ввода чисел не будет контролироваться. Ниже приведена программа на Ассемблере, решающая эту задачу.

```
include console.inc
N  equ 200000; const N=200000;
.data
   S     dd 0; искомая сумма  S=0
   Prig db "Вводите числа массива ",0
;  этот текст используется при вводе как массива X, так и Y
.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_very_good_program:
   outstr offset Prig; приглашение к вводу
   outcharln 'X';        массива X
   mov     ecx,N;   счётчик цикла
   mov  ❶ ebx,0;   индекс массива "i:=1"
L1: inint ❷ X[ebx]; ввод очередного X[i]
;  jc      Error;   при необходимости контроля ввода
   add     ebx,4;   это "i:=i+1"
   loop    L1;      цикл ввода X
   outstr offset Prig; приглашение к вводу
   outcharln 'Y';        массива Y
   mov     ecx,N;      счётчик цикла
   sub     ebx,ebx;    на 1 байт короче, чем ❶ mov ebx,0
L2: inint  Y[4*ebx]; на один байт длиннее, чем ❷ X[ebx] !
;  jc      Error;      при необходимости контроля ввода
   inc     ebx;        это "i:=i+1"
   loop    L2;       цикл ввода Y
   mov     ebx,offset X; ebx:=@X[1]
   mov     esi,offset Y+4*N-4; esi:=@Y[N]
;  можно и lea esi,Y+4*N-4; esi:=@Y[N]
   mov     ecx,N;
L3: mov     eax,[ebx]; первый сомножитель, это ebx↑=X[i]
   mul     dword ptr [esi]; умножение на esi↑=Y[N-i+1]
;  <edx:eax>=X[i]*Y[N-i+1]; CF=1, если edx<>0
   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. "j:=j-1"
   loop    L3;           цикл суммирования
   outwordln S,10,"Сумма="; это Write('S=',S:10)
   exit    0; Хороший конец программы
Err:
   outstr "Ошибка - большое значение! "
   exit    1; Плохой конец программы
;Error:
;  outstr "Ошибка - плохой ввод!"
;  exit    1; Плохой конец программы
   end begin_of_my_very_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 (это тип pointer в языках высокого уровня). Напомним, что адрес – это *смещение* элемента относительно *начала памяти* (в отличие от индекса элемента – это смещение от *начала массива*). Использование адресов элементов вместо их индексов позволяет работать с элементами массива, не зная *имени* этого массива. Это будет очень важно при программировании подпрограмм на Ассемблере, которые, как и подпрограммы па Паскале, не знают имён своих фактических параметров.

При записи команды умножения

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

нам необходимо *явно* задать размер второго сомножителя с помощью операции смены типа **ptr**, она *приказывает* считать, что операнд [ESI] задаёт адрес **dword**=**dd** именно двойного слова. Это необходимо, так как по виду операнда [ESI] компилятор Ассемблера сам не может сам «догадаться», что это элемент массива *размером в двойное слово* [vi] [см. сноску в конце главы].

В качестве альтернативы в нашей программе можно было бы использовать команду, где есть имя Y, явно задающее размер операнда:

```
mul Y[4*ecx-4]; X[i]*Y[N-i+1]
```

однако эта команда на 5 байт длиннее! В команде

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

Для прибавления очередного произведения, полученного на регистровой паре <EDX:EAX> к сумме S нам пришлось проводить суммирование ❶ по частям, по очереди прибавляя к S регистры EAX и EDX.

- Операция **type**.

На данные свои взирая объективно,
Задумал типы я и идеал создал;
Козьма Прутков.
«Безвыходное положение»

Для задания размера элемента массива использована операция **type**. Параметром этой операции обычно является некоторое имя, а значением операции **type** <операнд> является целое число – *тип* данного операнда. Операнд должен быть константным выражением, т.е. его значение должно быть известно на этапе компиляции.[1] Когда операнд задан именем областей памяти, то 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 имён процедур их тип – отрицательное число −252 (это значение служебных констант **near** и **proc**). Имена регистров r8, r16 и r32 имеют тип 1, 2 и 4 соответственно. Имена, описанные в директивах эквивалентности как макроконстанты (например, N **equ** 10), имеют тип ноль, это же число выдаётся при попытках использовать операцию **type** без имени (например, **type** [ebx] или **type** 333. Применение операции **type** к служебным именам, которые не являются регистрами, к именам макропеременных не целого типа (см. немного нижк), а также к неописанным именам приводит к ошибкам компиляции, например:

```
X  db ?
   mov eax,type add
Error A2008: syntax error
N  equ 10; Макроконстанта N
   mov eax,type N;  eax=0
   mov eax,type 10; eax=0
K=10; Макропеременная K (целого типа)
   mov eax,type K; eax=10
```

[1] Точнее, перед началом счёта, после работы редактора внешних связей и загрузчика, это подробно описано в разд. 9.2 и 9.3.

```
T   equ ABC; Макропеременная T (строкового типа)
    mov eax,type T; type ABC     ????
Error A2006: undefined symbol: ABC
    mov eax,type X; eax=1
    mov eax,type offset X; eax=4 (Длина адреса X)
    mov eax,type X[ebx]; eax=1
    mov eax,type offset X[ebx]; Адрес X[ebx] неизвестен!
Error A2098: invalid operand for OFFSET
```

6.7.4. Программирование циклов на Ассемблере

Иногда мне кажется, что единственным универсумом в программировании является цикл.

Алан Перлис,
первый лауреат премии Тьюринга

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

Пусть дана матрица знаковых двухбайтных целых чисел и надо найти сумму элементов, которые расположены в заданном столбце `j` этой матрицы. Для решения этой задачи на языке Free Pascal можно предложить следующую программу:

```
const N=20; M=30;
var X: array[1..N,1..M] of smallint;
    i,j: byte; S: longint=0;
begin {$R-}{$I+}{ Без «ручного» контроля }
  for i:=1 to N do for j:=1 to M do
    Read(X[i,j]);
  Read(j); {без проверки правильности j}
  for i:=1 to N do S:=S+X[i,j];
  Write(S)
end.
```

Для переноса программы с языка высокого уровня 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 главная 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, *линеаризировать* матрицу. Для большинства языков при этом сначала в секции данных размещается первая строка матрицы, сразу вслед на ней – вторая, и т.д. Отметим, что такая же линеаризация матрицы реализована и во входном потоке `input` (`stdin`). В нашем примере 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 програм-

мы 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:

Адрес(X[i,j]) = Адрес(X[1,1])+2*M*(i-1)+2*(j-1)

Эту формулу легко понять, учитывая, что каждая строка имеет длину `2*M` байт. Буквальное вычисление адресов элементов по приведённой выше формуле 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.). Получается, например, такая программа на Ассемблере:

```
N equ  20; Числовая макроконстанта N
M equ  30; Числовая макроконстанта M
.data?
X dw N*M dup (?); X[1..N,1..M] of smallint;
.data
S dd 0; Для суммы отводим переменную типа longint
j dd ?
.code
Start: ConsoleTitle "  Работа с матрицей"
; Ввод матрицы X
 ❶ mov   ecx,N*M; Ввод N*M элементов
 ❷ mov   ebx,offset X; ebx:=@X[1,1]
L: inint word ptr [ebx]; Read(ebx↑)
   add   ebx+2; На следующий элемент матрицы
   loop  L
   inint j
   mov   ebx,j
 ❸ lea   ebx,X[2*ebx-2]; ebx:=@X[1,j]
   mov   ecx,N; Число элементов в столбце
L: movsx eax,word ptr [ebx]; eax:=longint(ebx↑)
   add   S,eax; S:=S+X[i,j]
 ❹ add   ebx,2*N; j:=j+1
   loop  L1; Цикл по столбцу
   outintln S,,'Сумма='
   exit
   end Start
```

Приведённый пример хорошо иллюстрирует стиль мышления программиста на Ассемблере. Используя линеаризацию матрицы в памяти и в потоке `input` ввод всей матрицы производится в одном цикле на `N*M` повторений ❶. Для доступа к элементам матрицы применяются указатели (ссылочные переменные, значениями которых являются адреса) на регистрах, и используются операции над адресами в этих регистрах (так называемая адресная арифметика). Перед вводом мы установили указатель EBX на адрес ❷ `X[1,1]`, а перед суммированием элементов столбца мы установили указатель EBX на адрес ❸ `X[1,j]`. Для перехода к следующему элементу столбца мы прибавили к указателю EBX ❹ `2*N` байт. Получающиеся машинные 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).

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

Команда

`loopz/loope` L; это **`until`** (ECX=0) **`or`** (ZF=0)

выполняется по схеме

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

когда эта команда в конце цикла, то это как в Паскале

```
until (ECX=0) or (ZF=0)
```

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.

Вопросы и упражнения

Даже известное известно лишь немногим.

Аристотель, IV век до н.э.

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. Для чего в программе используется директива **include** ?
3. Как определить, где начинается и заканчивается секция программы ?
4. Можно ли использовать в программе две директивы начала секции данных **.data** ?
5. Можно ли выполнять команды из секции данных ?
6. В каких секциях могут располагаться области данных для хранения переменных ?
7. Почему реализованы две макрокоманды для вывода знаковых (**outint**) и беззнаковых (**outword**) чисел, в то время, как макрокоманда ввода целого числа всего одна (**inint**) ?
8. Какое преимущество имеет относительный переход перед абсолютным переходом ?
9. Для чего необходимы разные команды условного перехода после сравнения на больше или меньше знаковых и беззнаковых чисел ?
10. Что делать, если в команде цикла `loop L` необходимо передать управление на метку `L`, расположенную достаточно далеко от этой команды цикла ?
11. Чем отличается выполнение команды `inc op1` от выполнения команды `add op1,1` ?
12. Почему для деления числа в формате слова на маленькие числа часто необходимо использовать команду длинного, а не короткого деления ?
13. Почему при реализации цикла на Ассемблере, если это возможно, надо выбирать цикл с постусловием, а не цикл с предусловием ?
14. Когда перед операндом формата `m16` необходимо ставить явное задание длины операнда в виде `word ptr` ?
15. Как работает операция Ассемблера **type** ?

---

[i] Для продвинутых читателей. Таким образом, не требуется обязательного *выравнивания* данных в памяти, например, при чтении из памяти слова из двух байт, это слово может начинаться как с чётного, так и с нечётного адреса, а двойное слово (**dd**) не обязано располагаться с адреса, кратного четырём. Впрочем, иногда выравнивание всё-таки необходимо. Например, так называемой стековый кадр (о нём будет рассказано в другой главе) должен быть выровнен на границу четырёх байт, а данные в памяти, с которыми работают особые векторные регистры – на границу 16 байт. На процессорах Intel принудительный контроль выравнивания может быть включён установкой бита `AC` (Alignment Check) регистра `EFLAGS` и так называемого бита `AM` в управляющем регистре `CR0`, но обычно так не делается. Кроме того, работа с выровненными данными производится процессором более эффективно, поэтому, например, компилятор языка Free Pascal все скалярные переменные (если явно не сказано противное) выравнивает на границу 8 байт. Для выравнивания переменной в памяти про-

граммист на Ассемблере может перед её описанием вставить директиву **align** n, где n=1, 2, 4, 8 и т.д. байт, например:

```
        align 8
      X db ?
```

Отметим, что обязательное выравнивание необходимо при обращении к так называемым портам ввода/вывода, см. раздел 14.3.1.

Оптимизирующие компиляторы с языков высокого уровня часто ставят директивы выравнивания памяти перед началом процедур и критических по быстродействию циклов, например:

```
        align 16
      P proc
```

При этом «лишние» байты в секции кода заполняются пустыми командами **nop**, а в секциях данных – нулями. Такой приём позволяет гарантировать, что сразу после входа в процедуру или цикл несколько их первых команд уже будут находиться в кэш памяти (подробно об этом будет рассказано в другой главе).

[ii] Для продвинутых читателей. Кроме перечисленных, в этой таблице допустимы также операнды формата i16:m32 (i48) и m16:m32 (m48). Обычно такие операнды обозначаются как seg:offet, 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 и динамические связи между модулями по управлению с помощью внешних имен и входных точек, при этом команда формата **jmp** i48 формируется Ассемблером автоматически из команды **jmp far ptr** 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:offet=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 (учитываем, что адрес i48 представляется в памяти в перевёрнутом виде). Более "цивилизованный" (и более длинный) обходный путь заключается в записи адреса seg:offet в стек и выполнения затем команды дальнего возврата:

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

Отметим, что в 64-битном режиме команды **jmp** i16:i64 нет, есть только команда **jmp** m16:m64.

[iii] Для продвинутых читателей. При адресации относительно счётчика адреса для доступа к нужному месту памяти можно использовать маленькие смещения. Исходя из этого, в 64-битных ЭВМ адресацию (уже относительно 64-битного счётчика адреса RIP (RIP-relative addressing) можно применять и к командам, и к данным, при этом в качестве смещения тоже используются форматы i8 или i32, например:

```
.data
X  db ?
.code
...mov al,X; X=i32 (со знаком!)
```

Здесь у переменной X обычно получается не 64-битный, а только 8-битный или 32-битный адрес! Тогда, например, команда **mov** rax,m64, может иметь "странные" форматы r64,i32 или r64,i8. Доступ к секциям данных и кода по умолчанию выполняется относительно регистра RIP, это позволяет получать перемещаемый код, который правильно работает без настройки загрузчиком по конкретному адресу. Конечно, RIP является новым *базовым* регистром, но не регистром общего назначения, т.е. нельзя, скажем, написать команду **mov** rax,rip, однако можно **lea** rax,[rip+4*rbx] или **mov** rax,[rip]. Всё смешалось в доме Intel…🙁.

Отметим, однако, что относительный адрес *перехода* может вычислить сам компилятор с Ассемблера, так как точка перехода находится в той же секции, что и команда перехода. А вот относительный адрес переменной находится в *другой секции*, поэтому его может вычислить только редактор внешних связей, а при нестандартном размещении секций в памяти только загрузчик (см. главу 9).

[iv] Для продвинутых читателей. Флаг чётности PF (parity flag) равен единице, если в восьми младших битах результата содержится *чётное* число двоичных единиц. Этот флаг используется редко, в основном, как ни странно, для работы с *вещественными* числами.

Два целых числа (одинаковой длины) для нашего процессора всегда *сравнимы* (ordered) между собой, т.е. либо они равны, либо одно из них меньше другого (разумеется, учитывается, знаковые это числа или беззнаковые). При сравнении вещественных чисел ситуация сложнее, так как среди них могут быть особые «числа» (NaN), в этом случае говорят, что числа *несравнимы* (unordered) между собой.

Так вот, при сравнении вещественных чисел (командой **fcom**) устанавливается особый флаг сопроцессора C2=1, если вещественные числа *несравнимы* между собой. Все команды условных переходов, однако, смотрят только на флаги в регистре EFLAGS. Можно перенести флаги, выработанных в сопроцессоре, в регистр EFLAGS, это делается двумя «хитрыми» командами **fstsw** ax и **sahf**). При этом флаг C2 попадает именно в PF. После этого, несмотря на то что вещественные числа по своей сути являются *знаковыми*, нужно пользоваться командами условных переходов для *беззнаковых* целых чисел (**jne**, **ja**, **jbe** и т.д.). Флаг чётности PF может использоваться программистом для проверки того, что вещественные числа несравнимы между собой, например **jp** Real_Not_Compared.

Впрочем, есть и новая команда сравнения вещественных чисел **fcomi** (даёт исключение, если числа не сравнимы), а также команда сравнения **fucomi** с игнорированием «тихого» NaN, эти новые команды *дополнительно* устанавливают и флаги ZF, CF и PF, так что переносить в них из флагов сопроцессора C1, C2 и C3 не надо.

[v] Для продвинутых читателей. Команду **SETcc** иногда используют оптимизирующие компиляторы, чтобы избавиться при вычислении условных операторов от команд условных переходов (как ни странно это звучит). Например, пусть X знаковое число и надо закодировать условный оператор

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

Ниже показаны фрагменты на Ассемблере, полученные при программировании «в лоб» и оптимизирующим компилятором (опция оптимизации по скорости):

```
  mov Y,200h
  cmp X,0
  jl  L
  mov Y,300h
L:
```

```
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,0FFFFFC00h
;  if X<0 then ecx:=-100h else ecx:=0
add  ecx,300h
mov  Y,ecx
```

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.

А вот ещё одна реализация условного оператора оптимизирующим компилятором:

```
if X=0 then Y:=-99 else Y:=99
```

```
cmp X,1;      if X=0 then CF:=1 else CF:=0
sbb eax,eax; if X=0 then eax:=-1 else eax:=0
and eax,58;  if X=0 then eax:=-198 else eax:=0
add eax,99;  if X=0 then eax:=-99 else eax:=99
mov Y,eax
```

Компиляторы идут на всё, только бы избавиться от «плохих» команд условного перехода, например, так может транслироваться оператор X:=abs(X) (X формата **dd**, показан перевод «в лоб» и оптимизированные):

```
  cmp X,0
  jge L
  neg X
L:
```

```
mov eax,X
cdq ; edx:=-1 для X<0
xor eax,edx
sub eax,edx; +1 для X<0
mov X,eax
```

```
mov   eax,X
neg   eax; eax:=-X
test  X,80000000h
;if X<0 then X:=-X
movnz X,eax
```

Второй и третий варианты используют дополнительные регистры, однако выполняется быстрее. Разберитесь, как работает эти алгоритмы! Учтите, однако, что при наличии директивы {$R+} языка Free Pascal необ-

ходим контроль для случая, когда модуль числа X не существует. Например, для второго варианта после команды `sub eax,edx` компилятору придётся вставить:

```
jo Range_Checking_Error
```

Любопытно, что для *вещественных* значений реализация функции `abs(x)` примитивна:

```
x dd  ?; var x: single;
  and x,7FFFFFFFh; x:=abs(x)
```

А вот реализация оператора `Y:=sign(X)` (показан перевод «в лоб» и оптимизированный):

```
var X,Y: Longint;
Y:=-1; if X=0 then Y:=0 else if X>0 then Y:=1; {На Паскале}
Y:=-1; if X>=0 then Y:=ord(X<>0);              {На Паскале}
Y = (X<0) ? -1 : (X>0);                        // На языке С.
```

```
  mov Y,-1
  cmp X,0
  jl  L
  mov Y,0
  je  L
  mov Y,1
L:
```

```
xor   eax,eax
mov   ebx,1
mov   ecx,-1
cmp   X,0
cmovg eax,ebx
cmovl eax,ecx
mov   Y,eax
```

Ещё один пример реализации оператора `Z:=min(X,Y)` (целые знаковые числа формата **dd**):

```
Z:=X; if Y<X then Z:=Y; {На Паскале}
Z=(X<Y) ? X : Y;        // На С.
```

```
  mov eax,X
  cmp eax,Y
  mov Z,eax
  jl  L
  mov eax,Y
L:mov Z,eax
```

```
mov   eax,X
cmp   eax,Y
cmovg eax,Y
mov   Z,eax
```

```
mov eax,X
mov ebx,Y
sub ebx,eax
sbb ecx,ecx
and ecx,ebx
and eax,ecx
mov Z,eax
```

(Пример в третьем столбце предложен Агнером Фогом)

---

vi Для продвинутых читателей. Операция **ptr** является аналогом *явного* преобразования типа в языках высокого уровня, но работает принципиально по другому. Рассмотрим пример:

```
var x: byte; y longword;

   y:=x; {y:=longword x;}
{ это 1 байт x преобразуется в
 4 байта и записывается в y
 На Ассемблере это будет }
   movzx eax,y
   mov   y,eax
```

```
.data
x db ?
y dd ?

  mov y,dword ptr x
; это y:=4 байта из памяти,
; начиная с  x, т.е. это
; это y:=x,x+1,x+2,x+3
```

Операция **ptr** преобразует тип (т.е. длину!) операнда команды в памяти (но не на регистре!), перед **ptr** можно указывать любой тип (**byte**, **word**, **dword**, тип структуры и т.д.). Операция **ptr** не нужна, если тип одного операнда можно определить по типу другого операнда в команде, например, в команде

```
mov eax,dword ptr[ebx]
```

операция **ptr** не нужна (хотя и допускается). Когда студент использует здесь **ptr**, преподаватель осознаёт, что этот студент не понимает суть дела 🙁.

Заметим, что некоторые Ассемблеры (например, NASM) не требуют соответствие типов к операндов команды. Например, рассмотрим описание переменных

```
.data
  x db 123
  y dd 456
```

У нас для команды

```
mov eax,x;  Синтаксическая ошибка !
```

А в Ассемблере NASM командf

mov eax,[x]; это наше **mov** eax,x
просто считает на регистр EAX не только байт переменной x, но и следующие 3 байта из переменной y 🙁. Ясно, что это снижает надёжность программы.

6.8. Работа со стеком

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

Лао-цзы, V век до н.э.

Лучше сказать лишнее, чем не сказать необходимого.

Плиний Младший, I век 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 реализуется в виде области памяти, на текущую позицию (вершину) стека в этой области указывает регистр ESP (Extended Stack Pointer), таким образом, (какой-то) стек в машине есть *всегда*.

Так как за стеком закреплён «свой» регистр, то говорят, что в компьютере реализован именно аппаратный стек. Некоторые архитектуры ЭВМ (например, так называемые RISC процессоры, обычно используемые в смартфонах и планшетах) используют для работы со стеком обычные регистры общего назначения, в этом случае стек является «обычным» участком памяти, тогда говорят о программном стеке (сам компьютер о нём не знает).

В архитектуре процессоров x86 стек хранится в памяти в перевёрнутом виде: при увеличении размера стека его вершина смещается в область памяти с ме́ньшими адресами, а при чтении из стека – в область с бо́льшими адресами. В регистре ESP хранится смещение вершины стека от начала сегмента стека в памяти (но, как уже говорилось, в плоской модели памяти все сегменты совпадают и начинаются с нулевого адреса).

Стек есть аппаратная реализация абстрактной структуры данных *стек*, в языке машины предусмотрены специальные команды записи в стек и чтения из стека. В 32-битной архитектуре 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 на вершине стека, и читаться из стека первым, это 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 эти 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 двигается вверх (при записи в стек) и вниз (при чтении из стека).

В любой момент времени регистр ESP указывает на вершину стека – на *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. 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 первый байт за концом стека.

[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.

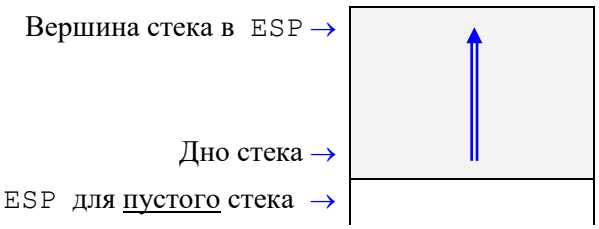


Рис. 6.3. Так будет изображаться стек.

То, что стек растёт вверх (к меньшим адресам) объясняется историческими причинами. На первых ЭВМ стек делил одну область памяти с кучей (heap), причём куча росла вниз, а стек вверх, когда они пересекались, фиксировалось исчерпание памяти.

Здесь необходимо также заметить, что в англоязычных книгах стек рисуют растущим сверху вниз, т.к. считается, что память надо изображать так, чтобы байты со старшими (бóльшими) адресами находились сверху (High Addresses – «верхние» адреса, как верхние этажи многоэтажного дома). Можно сказать, что мы представляем растущий стек как дерево (растёт снизу вверх), а в англоязычных странах как сосульку 🙂. Поэтому для нас, как и полагается, «дно» стека внизу, а «вершина» – сверху, а для них наоборот, «дно» сверху, а «вершина» снизу 🙄. Для компьютерной памяти такой «западный» подход приводит к любопытным эффектам: адреса памяти растут снизу-вверх, но во всех книгах адреса в изображаемых программах (да и массивах) растут сверху вниз 😕. В общем, всё не как у людей…

Программист может сам не описывать в своей программе секцию стека, она создаётся 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. Как задать на Ассемблере переменным в стеке имена будет рассказано ниже, при описании процедур.

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. Можно также считать, что загрузчик размещает программу в памяти и помещает в регистр `EIP` адрес команды, помеченной меткой (`Start`), указанной в директиве **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.[1] При работе программы стек может расти (т.е. значение регистра `ESP` будет уменьшаться). Сначала программе даётся совсем маленький стек (обычно 4 так называемые страницы памяти по 4096 байт каждая). При переполнении стека (выходе значения регистра `ESP` за верхнюю границу), операционная система автоматически увеличивает размер стека (в пределах отведённой под стек памяти).[i] [см. сноску в конце главы]

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

push `op1;` это в учебной УМ-0 `ВСТЕК op1`

где операнд `op1` может иметь форматы `r32`, `m32`, `i8` и `i32`.[2] Команда записывает в стек свой операнд, для операндов `r16` и `m16` это 2 байта, а для операндов `r32` и `m32` это 4 байта. При записи в стек операндов `i8` и `i32`, они *знаково* расширяются до 32 бит. Эта команда выполняется по такому алгоритму:

```
case op1 of
  r16,m16:begin ESP:=(ESP-2) mod 2^32; <ESP>:=op1 end;
  r32,m32:begin ESP:=(ESP-4) mod 2^32; <ESP>:=op1 end;
```

[1] Это упрощённое (и, следовательно, не совсем правильное) изложение запуска программы на счёт. На самом деле загрузчик заполняет только специальную область сохранения `TSS` (там, в частности, находятся и значения регистров) и передаёт программу на выполнение так называемому диспетчеру процессов, более подробно это будет описано в разд. 9.3.

[2] В стек можно записывать и содержимое 16-битных сегментных регистров `ES`, `DS`, `SS`, `FS`, `GS` и `CS`, они при этом расширяются нулями до 32 бит, мы такие команды использовать не будем.

```
  i8,i32:
    begin ESP:=(ESP-4) mod 2^32; <ESP>:=Longint(op1) end;
  ES,DS,SS,FS,GS,CS:
    begin ESP:=(ESP-4) mod 2^32; <ESP>:=Longword(op1) end;
end
```

Например (вспомним форматы команд ☺):

```
push ebx;    Длина команды 1 байт
push bx;     Длина команды 2 байта
;      66h   push ebx
push 100;    Длина команды 2 байта
push -1;     Длина команды 2 байта
push 1000;   Длина команды 5 байт
```

Как видим, смысл отдельного операнда `i8` в том, что команда получается короче, чем для операнда `i32`. Особым является случай, когда при вычислении значения `op1` тоже используется регистр `ESP`. Например, команда

push esp

выполняются по схеме

<ESP-4> := ESP; ESP := (ESP-4) **mod** 2^32

Как видно, значение регистра `ESP` записывается в стек до изменения значения этого регистра. Команда

pop op1; это в учебной УМ-0 ИЗСТЕКА 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.[1] Эта команда выполняется по правилу:

```
case op1 of
  r16,m16:begin op1:=<ESP>; ESP:=(ESP+2) mod 2^32 end;
  r34,m34,ES,DS,SS,FS,GS,CS:
    begin op1:=<ESP>; ESP:=(ESP-4) mod 2^32 end;
end
```

Команды **push** и **pop** являются избыточными, их можно реализовать другими командами, например:

push eax → **sub** esp,4; **mov** [esp],eax
pop eax → **mov** eax,[esp]; **add** esp,4

Современное программное обеспечение требует, чтобы вершина стека всегда располагалась с адреса, по крайней мере кратного 4-м, 8-ми (а во многих случаев даже 16-ти байт). Это связано с тем, что в стеке могут храниться и переменные длиной 16 байт (это регистры `XMM`). Так что надо быть аккуратными при использовании «обычных» **push** и **pop**.

Все системные функции Windows (системные вызовы), например, которые используются в макрокомандах, крайне «болезненно» реагируют на то, что значение регистра `ESP` не кратно 4, чаще всего они не работают. Поэтому при программировании на Ассемблере лучше придерживаться правила, по которому в стек пишутся и из стека читаются только двойные слова (**dd**). Аналогично, для 64-битных ЭВМ стек выравнивается на границу 32-х байт. Это тем более обидно, так как сам процессор спокойно работает с не выровненным стеком.

Команды без явного параметра

pushf и **pushfd**

---

[1] В 64-битном режиме в командах **push** и **pop** добавляются операнды `i16` и `m16`, а операнды `ES`, `DS`, `SS` и `CS` запрещены. Операнды `i8` и `i16` обычно преобразуются в `int64`. В стеке могут храниться и переменные длиной 16 байт (регистры `XMM`), поэтому значение `ESP` должно быть кратно 16, так что надо быть аккуратными при использовании «обычных» **push** и **pop**, выравнивая адрес `ESP` на границе 16 байт.

записывает в стек соответственно регистр флагов Longword(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. Эти команды удобны для сохранения в стеке и восстановления значения регистра флагов, кроме того, так можно, скажем, копировать EFLAGS в регистр общего назначения: [ii] [см. сноску в конце главы]

```
pushfd
pop eax; eax:=EFLAGS
```

Как уже упоминалось, область стека можно использовать и как обычную память, например, с помощью регистра ESP как базового

```
add [esp],eax; <esp>:=<esp>+eax
```

Команды

pusha и **pushad** [1]

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** [1]

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 64-битном режиме из-за возросшего числа регистров общего назначения и векторных регистров эти команды не работают 🙁.

В качестве примера использования стека рассмотрим программу для решения следующей задачи. Необходимо вводить целые *знаковые* числа формата двойного слова **dd** до тех пор, пока не будет введено число ноль (признак конца ввода). Затем следует вывести *в обратном порядке* те из введённых чисел, которые больше нуля. Будем считать, что количество таких (положительных) чисел не более 100000, при достижении этого предела будем принудительно прекращать ввод и выдавать результат. Ниже приведено возможное решение этой задачи.

```
include console.inc
N equ 100000; макс. число положительных чисел
.code
My program_start:
   outstrln 'Вводите целые числа до нуля:'
   sub   ecx,ecx; это лучше, чем mov ecx,0 [2]
L: inint eax
;  jc    Error;    при необходимости контроля ввода
   cmp   eax,0;    конец ввода ? Лучше or eax,eax
   jl    L;        число<0,  на продолжение ввода
   je    Pech;     на вывод результата
   push  eax;      запись положительного числа в стек
   inc   ecx;      счётчик количества чисел в стеке
   cmp   ecx,N;    макс. число положительных чисел ?
   jb    L;        на продолжение ввода
   outstrln "Много чисел больше 0!"
Pech:
   cmp   ecx,0;    лучше or ecx,ecx
```

---

[1] Это одна и та же команда, просто для команд **pusha** и **popa** впереди ставится уже хорошо известная нам команда-префекс смены длины операнда 66h.

[2] Для обнуления регистра можно использовать и команду **xor** ecx,ecx, о которой будет рассказано в главе, посвященной логическим командам. К сожалению, команды **sub** и **xor** при обнулении регистра «портят» флаги, если это в данном месте программы недопустимо, то приходится использовать команду **mov**, хотя она медленная и более длинная.

```
    jnz    L1;       в стеке есть числа
    outstrln 'Нет положительных чисел'
    exit
L1: outstrln 'Числа в обратном порядке:'
L2: pop    eax
    outint eax,10; Write(eax:10)
    dec    ecx
    jnz    L2
    exit
    end My_program_start
```

Заметим, что в этой программе нет собственно переменных, а только строковые константы, поэтому не описана отдельная секция данных. Впрочем, эта секция будет определена и заполнена строками (`'Вводите числа до нуля:'` и т.д.) при обработке соответствующих макрокоманд.

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 и возврата из подпрограммы

Мудр тот, кто знает не многое, а нужное.

Эсхил, отец европейской трагедии

Команды вызова подпрограмм (напомним, что это обобщённое название процедур и функций) 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 запоминают в стеке адрес следующей за ними команды. Таким образом, обеспечивается *возможность* возврата в точку программы, следующую непосредственно за командой вызова подпрограмы. На языке Ассемблера эти команды имеют следующий вид:

```
      call op1; op1=r32,m32,i32 [2]
EIP → <следующая команда>
```

Вызов подпрограмы 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.

В архитектуре процессоров Intel счётчик адреса `EIP` почти всегда указывает на начало следующей команды. Это не так только в случае возникновения некоторых исключительных (аварийных) ситуаций, после которых счётчик адреса указывает на команду, вызвавшую ошибку, это позволяет при необходимости *повторить* данную команду.

Интересно, что в процессорах ARM (они используются в большинстве смартфонов и планшетов) счётчик адреса обычно указывает не на следующую команду, а «через одну». Длина команды там обычно 4 байта, так что счётчик стоит на «плюс 8» (а иногда и на «плюс 12»). Это позволяет функции возвращать результат её работы не на регистре (см. далее стандартные соглашения о связях), а записывать на место «пустой» команды (или двух команд), стоящей сразу за командой **`call`**. На наших компьютерах это будет сделать трудно, так как секция кода обычно закрыта на запись.

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

---

[1] В некоторых книгах по Ассемблеру такие команды называют «командами перехода с возвратом», что, конечно, неверно, так как сами эти команды никакого «возврата» не производят.

[2] Возможны ещё операнды форматов `m16:16` и `m16:32`, они определяют вызов так называемых дальних процедур (см. разд. 6.14), мы их использовать не будем.

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

Как видно, у этой команды единственный параметр формата i16. Команда возврата выполняется по схеме:

Изстека(EIP); ESP := (ESP + i16) **mod** 2^{32}

Здесь, по аналогии с командой вызова процедуры, запись Изстека(EIP) обозначает действие «считать из стека двойное слово в регистр EIP», такой «отдельной» машинной команды нет. Беззнаковый параметр i16 предназначен для удаления из стека i16 байт, обычно он используется для очистки стека от фактических параметров, о чём будет говориться далее [iii] [см. сноску в конце главы].

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

6.10. Программирование подпрограм на Ассемблере

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

Алан Перлис,
первый лауреат премии Тьюринга

Функции используются для наведения порядка в хаосе алгоритмов.

Бьёрн Строуструп

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

```
<имя процедуры> proc [<спецификации>]
     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.

В отличие от языков высокого уровня, где процедуры и функции, как правила, описываются до раздела операторов, в Ассемблере описание процедуры можно располагать в любом месте, но *только* в секции кода **.code**. Имя процедуры считается *меткой команды* (хоть и описывается без двоеточия).

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 одной процедуры 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 описанные в процедуре 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::
Local_Label:
N   equ 100; Имя N видно извне процедуры
    inc ebx
MyProc endp
    jmp Local_Label; ОШИБКА, не видно метку
    jmp Global_Label; Видно метку
    mov eax,N; Видно имя N
```

Иногда программисту нужно сделать, чтобы процедура имела несколько имён (одно основное и синонимы). Для этих целей в Ассемблере можно использовать директиву:

alias <синоним имени>=<имя процедуры>

Например, **alias** <YourProc>=<MyProc>

Глобальную метку можно поставить и *вне* тела процедуры, тогда она ничем не отличается от обычной метки.[1] Обратите также внимание, что *изнутри* процедуры все внешние (глобальные) метки видны, и при конфликте имён, к сожалению, предпочтение отдаётся *глобальной* метке, например:

```
MyProc proc
jmp L;  Переход на глобальную L:
X    dd  ?; Имя X снаружи видно
L: ;        Эта метка снаружи не видна
MyProc endp
. . .
L:  ;       Будет переход сюда  ! ! !
```

Создаётся впечатление, что программисты, создавшие компилятор MASM, 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
L:
```

В Ассемблере разрешены также так называемые анонимные метки **@@:**, такие метки могут встречаться в программе в любом количестве. Любая команда перехода **jxx** @F производит переход на ближайшую метку **@@:** *вниз* по тексту программы (Forward – вперёд), а команда **jxx** @B, наоборот, на ближайшую метку **@@:** *вверх* (Back – назад).

По замыслу авторов языка Ассемблера, это должно резко сократить количество «именованных» меток в программе. 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>
EIP →
```

Учтите, что, как уже говорилось, имя процедуры имеет тип метки команды, хотя за ним и не стоит двоеточие. Вызов процедуры обычно производится командой **call**, а возврат из процедуры – командой **ret**. Отметим, однако, что команду **ret** можно использовать и *вне* тела процедуры.

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

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

```
    mov  ax,1
P   proc
    add  ax,ax
P   endp
    sub  ax,1
    call P
```

Этот фрагмент полностью эквивалентен таким командам:

---

[1] Можно приписать всем меткам внутри процедур глобальную область видимости, задав директиву **option NoScoped**, но лучше этого не делать, сохранив в процедуре хоть какую-то локальность имён.

```
   mov  ax,1
P: add  ax,ax
   sub  ax,1
   call P
```

Другими словами, описание процедуры на Ассемблере может встретиться в любом месте секции команд, 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:

```
X  dd   L
L: mov  ax,1; процедура ???
      . . .
   call L
   mov  eax,offset L
   call eax; тоже call L
   call X;   тоже call L
```

Здесь вместо имени процедуры в команде **call** указаны обычная метка, а также регистр и переменная, содержащие адрес метки L. 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 (**dd**), 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. выдавать 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 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 семантическое. Другими словами то, что это функция, а не процедура, знает программист, но не компилятор Ассемблера. Как мы уже говорили далее будет использоваться обобщенный термин подпрограмма.

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

Соглашение о связях между основной программой и подпрограммой включает в себя место расположения и способ доступа к параметрам, способ вызова и способ возврата результата работы (для функции) и некоторую другую информацию. Так, для нашего последнего примера «договоримся» с подпрограммой, что адрес (не индекс!) первого элемента 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<>0 – количество элементов, eax – ответ (сумма)
   sub  eax,eax;     сумма:=0
@@:add  eax,[ebx];  прибавление текущего элемента
   add  ebx,4;       на адрес след. элемента "i:=i+1"
   loop @B;          на ближайшую @@: вверх
   ret
Summa endp
Start:
; здесь команды для ввода массивов X и Y
   mov  ebx,offset X; ❶ адрес начала X: ebx:=@X[1]
   mov  ecx,100;      число элементов X
   call Summa
; EIP →
   mov  Sum,eax;      сумма массива X
   lea  ebx,Y;        ❷ адрес начала Y: ebx:=@Y[1]
   mov  ecx,200;      число элементов Y
   call Summa
   add  Sum,eax;      сумма элементов массивов X и Y
   outintln Sum,,"Сумма="
   exit
   end Start
```

Обратите внимание на два способа записи адреса начала массива в регистр. В строке ❶ показан способ взятия адреса массива X с помощью операции **offset** (offset – смещение от начала памяти):

mov ebx,**offset** X; ❶ ebx:=адрес начала X

А в строке адрес массива берётся уже известной Вам командой **lea**:

lea ebx,X; ❷ ebx:=адрес начала X

Учтите, что способ ❷ на один байт длиннее.[1] Здесь часто делают ошибку, используя для взятия адреса команду:

mov ebx,X; ebx:=X[1] это не адрес, а сам первый элемент массива X

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); {$I+}{$R-}
var X: array[1..100] of Longint;
    Y: array[1..200] of Longint;
    ebx: ↑Longint; Sum,eax: Longint;
    ecx: Longword;
procedure Summa; {$goto+} {разрешить goto}
    label L;
begin
    eax:=0;
```

[1] Для продвинутых читателей. Любопытно, но в 64-битном режиме, наоборот, команда **lea** rbx,X использует 32-битную относительную адресацию по регистру RIP и короче, чем команда **mov** rbx,**offset** X с 64-битным адресом.

```
L:  eax:=eax+ebx↑;
    ebx:=ebx+4; dec(ecx);
    if ecx<>0 then goto L
end {Summa};
begin { Ввод массивов X и Y }
    ecx:=100; ebx:=@X[1];
    Summa; Sum:=eax;
    ecx:=200; ebx:=@Y[1];
    Summa; Sum:=Sum+eax; Writeln(Sum)
end.
```

Как видно, в этой программе используется плохой стиль программирования, так как неявными параметрами подпрограммы являются глобальные переменные, т.е. полезный механизм передачи параметров Паскаля просто не используется. В то же время именно хорошо продуманный аппарат формальных и фактических параметров делает подпрограммы таким гибким, эффективным и надежным механизмом в языках программирования высокого уровня. Если с самого начала решать задачу суммирования массивов на языке Free Pascal, а не на Ассемблере, надо бы, конечно, написать, например, такую программу:

```
program A(input,output); {$I+}{$R-}
  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 массивов X[1..100] и Y[1..200] }
  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 подпрограммами и на Ассемблере, однако для этого понадобятся другие, более сложные, соглашения о связях между подпрограммой и основной программой. При работе с подпрограммами на языке Ассемблера будут использоваться так называемые стандартные соглашения о связях.

Если Вы хорошо изучили программирование на языках высокого уровня, то должны знать, что грамотно написанная подпрограмма получает все свои входные данные как фактические параметры, возвращает результаты через фактические параметры (переданные по ссылке) и не использует внутри себя глобальных имён переменных и констант. Такого же стиля работы с подпрограммами надо, по возможности, придерживаться и при программировании на Ассемблере.

6.11. Стандартные соглашения о связях

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

Вадим Синявский

Сначала поймём необходимость существования некоторых *стандартных соглашений о связях* между процедурой и вызывающей её основной программой.[1] Действительно, иногда программист просто не сможет, скажем, «договориться» с процедурой, как она должна принимать от него свои параметры. В качестве первого примера можно привести так называемые *библиотеки стандартных подпрограмм*. В этих библиотеках собраны процедуры и функции, реализующие полезные алгоритмы для некоторой предметной области (например, для работы с матрицами), так что программист

[1] Для языков низкого уровня такие соглашения по-английски называют ABI (Application Binary Interface).

может внутри своей программы вызывать нужные ему алгоритмы из такой библиотеки. Библиотеки стандартных подпрограмм обычно поставляются в виде набора так называемых *объектных модулей*, что практически исключает возможность вносить какие-либо изменения в их исходный текст (с объектными модулями Вы познакомитесь в Главе 9). Таким образом, получается, что взаимодействовать с подпрограммами 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, С, Питоне и т.д.), а подпрограмма – на Ассемблере. Вспомним, что когда говорилось об областях применения Ассемблера, то одной из таких областей и было написание подпрограмм, которые вызываются из программ на языках высокого уровня. Например, для языка Free Pascal такая, как говорят, *внешняя*, функция может быть описана в программе на Паскале следующим образом:

```
function Summa(var X: Mas; n: Longword): Longint; external;
```

Служебное слово **external** является указанием на то, что эта функция не описана в данном программном модуле и Паскаль-машина должна вызвать эту *внешнюю* функцию, написанную, вообще говоря, на другом языке, и как-то передать ей параметры. Если программист пишет эту функцию на Ассемблере, то он конечно никак не может «договориться» с компилятором Паскаля, как он хочет получать параметры и возвращать результат работы функции.[1]

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

1. Соглашения об именах

   Необходимо договориться о том, *каким* вызывающая программа «видит» имя (внешней для неё) подпрограммы. Внутри одного программного модуля этой проблемы не существует, так как и вызывающая программа и подпрограмма находятся для компилятора в одной *области видимости*. Проблема возникает, если вызывающая программа и подпрограмма находятся в *разных* программных модулях и непосредственно не видят друг друга. Разные языки программирования (точнее, разные системы программирования) имеют свои правила о том, как имя подпрограммы 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`. Далее надо договориться, предполагается ли неразличимость больших и малых букв имён во «внешнем мире». Подробно об этом мы будет говориться в главе 9, посвященной модульному программированию.

2. Соглашения о передаче параметров и возврате значения функции

   Далее, необходимо договориться о том, как *фактические* параметры из главной программы передаются в подпрограмму на место *формальных* параметров (parameter transfer methods). Как известно, в языке Паскаль параметры можно передавать в подпрограмму по значению (pass by value) и по ссылке (pass by reference), это верно и для многих других языков высокого уровня. 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 в языке машины 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 (до 32 бит) параметры-значения передаются в формате **dd**, расширяя, при необходимости, их до размера 32 бит. Будет ли это расширение знаковым или

---

[1] На языке Free Pascal программист должен ещё иногда как-то указать, в каком файде находится объектный модуль с этой внешней процедурой, это можно сделать, например, с помощью директивы `{$Link <имя файла>}` (имя `Link` можно сократить до одной буквы `L`).

беззнаковым, естественно, зависит от типа этого параметра.[1] Для передачи по значению больших *параметров-значений* (например, строк символов и массивов) в системах программирования предусматриваются специальные правила.

Наиболее быстрой является передача параметров через регистры, как это было сделано выше в функции `Summa`. Регистров, однако, не так много, и к тому же они интенсивно используются при работе программы, поэтому обычно приходится первые 3 или 4 параметра передавать через регистры, а остальные передавать через стек.[2] 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;
```

В этом случае *адрес* `X` передаётся в процедуру на регистре `EAX`, а *значения* `Y` и `Z` соответственно на регистрах `EDX` и `ECX`. Для передачи вещественных параметров-значений дополнительно используются вещественные регистры. Для языка C вместо ключевого слова **`register`** при передаче параметров через регистры используется похожий способ передачи с именем **`fastcall`** (правда, можно использовать только два регистра: `ECX` и `EDX`).

Регистров, однако, мало, так что в общем случае этот способ имеет свои недостатки. Более универсальным является другой способ передачи, когда фактические параметры перед вызовом подпрограммы все записываются в стек. Для передачи вещественных параметров-значений дополнительно используются вещественные регистры `XMM`.

Итак, при передаче параметра в стеке *по значению* в стек записывается само это значение, при необходимости преобразованное в формат **`dd`**, а в случае передачи параметра *по ссылке* в стек записывается адрес начала фактического параметра.

В объектно-ориентированных языках при вызове подпрограмм, описанных внутри объекта, передаётся дополнительный (неявный) параметр **`this`**, содержащий ссылку на текущий объект.

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 большинстве языков это обратный порядок, однако, например, в языке Фортран – *прямой*.

Обратный порядок передачи параметров применяется и в Windows при использовании так называемых системных вызовов (видимо, это отзвук того, что бóльшая часть Windows написана на языке C). Впрочем, чтобы не путать программистов на Ассемблере, обычно при записи системного вызова используется стандартная макрокоманда **`invoke`**, параметры в ней идут в обычном порядке, а уже она сама меняет их порядок.

Прямой способ передачи параметров неудобен для подпрограмм с *переменным* числом фактических параметров (например, это процедура `write` Паскаля или функция `printf` языка C). В этом случае при изменении числа параметров все их адреса в стеке меняются! В то же время при обратном способе передачи параметров первый параметр всегда находится в стеке на одном и том же месте – сразу после адреса возврата. Обычно по значению первого параметра процедура может определить, сколько ещё параметров находится в стеке, например, для функции вывода языка C:

```
printf("Hi %c %d %s",'c',10,"there!");
```

Здесь число символов `%` в первом параметре задаёт (но не всегда!) число остальных параметров.

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);
```

---

[1] Отметим, что многие компиляторы с языков высокого уровня могут вообще оставлять в старших битах расширенного параметра (помним о перевёрнутом представлении в памяти!) просто «мусор».

[2] Сейчас 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.

указание

```
pascal; external;
```

задаёт *прямой* способ передачи параметров, а указание

```
stdcall; external;
```

задаёт *обратный* способ передачи параметров во внешнюю процедуру.

Далее, надо договориться о том, как функция будет возвращать свой результат в вызвавшую её программу. Обычно функция возвращает своё значение в регистрах AL, AX, EAX или в паре регистров <EDX:EAX>, в зависимости от величины (типа) этого значения (для значений AL и AX старшая часть регистра EAX, вообще говоря, не определена). Для возврата целых значений, превышающих учетверённое слово (**dq**), устанавливаются специальные соглашения. Вещественный результат возвращается на регистре st(0) как тип Extended.[1]

> При работе со стеком действует жёсткое правило: после возврата из подпрограммы и перед возобновлением счёта главной программы стек должен находится точно в таком же состоянии, как он был перед вызовом подпрограммы.

Напомним, что в языках высокого уровня *формальные* параметры, в которые передаются соответствующие им *фактические* параметры, являются *локальными* переменными подпрограммы, и, как любые локальные переменные, должны быть уничтожены при выходе из подпрограммы.

Исходя из этого надо договориться, кто будет очищать стек: подпрограмма перед возвратом, или главная программа после возврата. Мы будем предполагать, что перед возвратом из подпрограммы она обычно сама очищает стек от всех фактических параметров.

Исключением является особый способ передачи параметров, принятый в языке C, 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). Такой способ полезен только при передаче *переменного* числа фактических параметров, в основном он применяется в языке C. В этом случае обычно первый параметр содержит информацию об общем числе параметров, а при соглашении **pascal** положение первого параметра в стеке *неизвестно*! Плохо в этом случае то, что удаление параметров из стека приходится выполнять не в одном месте (в конце процедуры), а в *каждой* точке возврата из процедуры, что увеличивает размер кода (в программе может быть много вызовов одной и той же процедуры).

3. Соглашения о локальных переменных

Если в подпрограмме 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, размер памяти под локальную переменную округляется до двойного слова. Перед возвратом из подпрограмми стек очищается от всех локальных переменных, напомним, что в языках высокого уровня это стандартное действие при выходе из подпрограммы (в общем случае из блока).

4. Соглашения об использовании регистров

Часть регистров может быть объявлены как непостоянные (volatile), подпрограмма может изменять их без сохранения и восстановления. Остальные регистры считаются постоянными (immutable, nonvolatile), их значения не должны изменяться после возврата из подпрограммы. Таким образом, если в подпрограмме необходимо использовать постоянные регистры, то необходимо сначала их где-то (обычно в стеке) запомнить, а потом восстановить. Для функции, естест-

---

[1] В 64-битном режиме функция может также возвращать вещественное значение на регистре XMM0 (128 бит).

венно, не запоминаются и не восстанавливаются регистр(ы), на котором(ых) возвращается результат её работы. Обычно также принято не запоминать и не восстанавливать регистр флагов, регистры для работы с вещественными числами и векторные регистры.

Точнее, все вещественные и векторные регистры должны очищаться перед вызовом (если их предполагается использовать в подпрограмме) и перед возвратом (если они менялись в процедуре). В регистре флагов при входе и выходе рекомендуется устанавливать флаг `DF=0`, об этом флаге будет говорится далее. Напомним, что наши макрокоманды из пакета `io.inc` более «строгие», они сохраняют все регистры общего назначения и регистр флагов.

По стандартному соглашению о связях системные вызовы функций API Microsoft Windows, как уже говорилось, должны запоминать и восстанавливать регистры `ESI`, `EDI`, `EBP` и `EBX`, на `EAX` возвращается результат функции, сохранность остальных регистров (`ECX` и `EDX`) не гарантируется. Такой выбор не восстанавливаемых регистров легко понять, `EAX` и `EDX` используются для возврата значений функции и в командах умножения, а и `ECX` как счётчик цикла, их «портят» большинство подпрограмм. По этим же соображениям эти регистры выбраны и для передачи в подпрограммы параметров.

В программировании есть понятие состояние программы, это значения всех существующих в данный момент переменных (в оперативной и внешней памяти программы) и указатель текущей позиции выполнения алгоритма (регистр адреса `RIP`). Все современные архитектуры ЭВМ обеспечивают безусловную защиту оперативной и внешней памяти одной программы от случайного или предумышленного использования и изменения другими программами.

Сложнее обстоит дело с регистровой памятью, она принципиально общая для всех выполняющихся программ. Регистров много, это регистры общего назначения, вещественные, векторные и служебные (управляющие, отладочные и т.д.) регистры. Соответственно, вводится понятие контекста программы, это значения всех регистров, которые определяют текущее состояние вычислительного процесса. Если спасти контекст программы, то можно надолго прервать её выполнение, а затем продолжить счёт программы с прерванного места (если, конечно, за это время не «испортилась» оперативная и внешняя память программы).

Далее, так как подпрограмма рассматривается как самостоятельная часть алгоритма, то можно ввести понятие контекста подпрограммы. Таким образом, необходимо обеспечить сохранность контекста главной программы после возврата из подпрограммы.

В общем случае спасение и восстановление контекста возлагается как на главную программу, так и на подпрограмму. Участок стека, в котором при *каждом* вызове подпрограммы размещаются контекст главной программы, фактические параметры, локальные переменные и адрес возврата подпрограммы, называется стековым кадром (stack frame).[1]

Большинство подпрограмм требует стековый кадр для работы (они называются frame function). Стекового кадра могут не иметь только так называемые *простые* (конечные) подпрограммы (final function, leaf function), которые не «портят» постоянные регистры, не вызывают другие подпрограммы и не порождают в стеке локальных переменных. Для доступа к параметрам в стеке такая функция использует в качестве базового не регистр `EBP`, а регистр `ESP` (об этом будет рассказано далее). Правда, команды с адресацией `[ESP+i8]` на один байт длиннее, чем `[EBP+i8]`, но при этом регистр `EBP` освобождается для других операций.

Такие подпрограммы используются редко, так как они обычно маленькие по размеру и вместо их вызова можно просто подставить их тело на место вызова, использую технику так называемых **`inline`** подпрограмм. Кроме того, с такими подпрограммами плохо работают отладчики.

Стековый кадр начинает строить основная программа перед каждым вызовом подпрограммы, помещая туда (т.е. записывая в стек) фактические параметры. Затем команда **`call`** помещает в стек адрес возврата (return address – 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.

---

[1] В англоязычной литературе для этого термина иногда используется малопонятный синоним activation record (запись активации 😳).

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 (кроме, как уже говорилось, соглашения **cdecl**).

> Наличие у каждой системы программирования своих, немного разных, стандартных соглашений о связях, по существу означает, что по настоящему «стандартных» соглашений нет. Это, конечно, затрудняет написание универсального (как говоря, портабельного) программного обеспечения. И не случайно в 64-битной ОС Windows осталось только одно такое соглашения под названием **vectorcall** (см. см. разд. 6.12). В то же время, к сожалению, в 64-битных ОС семейства Linux соглашение о связях тоже одно, но уже своё… 😕

6.12. Соглашение о связях stdcall

Это ещё не конец. Это даже не начало конца. Но это, возможно, конец начала.
Уинстон Черчилль, 1942 год,
это Антиметáбола 🙄

В примерах этой книге 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.

Итак, вспомним нашу последнюю «хорошую» программу суммирования массивов на языке Free Pascal:

```
program A(input,output); {$I+}{$R-}
  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 массивов X[1..100] и Y[1..200] }
  Sum:=Summa(X,100)+Summa(Y,200); Writeln(Sum)
end.
```

Перепишем эту программу на Ассемблере с использованием стандартного соглашения **stdcall**. Описание функции Summa 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(?); X:array[1..100] of Longint;
   Y   dd 200 dup(?); Y:array[1..200] of Longint;
Sum dd ?
.code
Summa proc
; стандартные соглашение о связях stdcall
   push ebp
 ❶ mov  ebp,esp; ebp – база стекового кадра
 ❸ sub  esp,4;    порождение локальной переменной S=?
```

```
S   equ  dword ptr [ebp-4] ❷
;   S  будет именем локальной переменной в стеке [1]
    push ebx;        запоминание используемых
    push ecx;        регистров ebx и ecx
    mov  ecx,[ebp+12]; ecx:=длина массива N
    mov  ebx,[ebp+8];  ebx:=адрес первого элемента
    mov  S,0;          сумма:=0, это
;   mov  dword ptr [ebp-4],0
@@:mov  eax,[ebx];     нельзя [add S,[ebx]] – нет формата
    add  S,eax;        {$R-} т.е. не смотрю на OF
    add  ebx,4;        "i:=i+1"
    loop @B;           на ближайшую @@ вверх
    mov  eax,S;        результат функции на eax
  ❹ pop  ecx
    pop  ebx;          восстановление регистров ecx,ebx
; ↓ уничтожение локальной переменной S [iv] [см. сноску в конце главы]
    mov  esp,ebp
    pop  ebp;          восстановление регистра ebp
    ret  2*4;          возврат с очисткой стека от 2-х параметров
Summa endp
; основная программа
Start:
; здесь команды для ввода массивов X и Y
    push 100;        ❺ второй фактический параметр
    push offset X;   ❺ первый фактический параметр
    call Summa
; здесь стековый кадр полностью уничтожен
    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 ❺ в обратном порядке.

Для записи в стек параметров современные компиляторы обычно используют другой метод:

```
sub esp,2*4;                  место в стеке под два параметра
mov [esp],offset X;           вместо push offset X
mov dword ptr [esp+4],100;    вместо push 100
```

Это позволяет записывать параметры в стек в естественном для человека прямом порядке (сначала первый параметр, затем второй и т.д.). Заметим, что такие команды лучше выполняются на компьютерном конвейере, так как вершина стека изменяется только один раз (для конвейера нет зависимости по данным, эта тема будет изучаться в другой главе). Кроме того, этот способ позволяет записывать в стек большие параметры, например, значения векторных регистров:

```
sub esp,16;           16 байт под регистр xmm0
vmovdqa [esp],xmm0;   128-битный регистр xmm0
```

[1] К сожалению, это имя (в отличие от самой переменной) не локальное в процедуре, т.е. будет видимо извне её.

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 EBP).

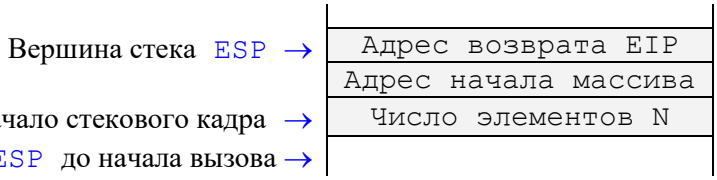


Рис. 6.4. Вид стекового кадра при входе в функцию Summa.

Отметим далее то значение, которое отводится индексному регистру EBP при работе со стеком. Этот регистр используется в процедуре как *база* стекового кадра для доступа к параметрам и локальным переменным процедуры или функции. Так и было сделано в этой программе, когда командой ❶ регистр EBP (по сути, это ссылочная переменная в смысле Паскаля) был поставлен примерно на середину стекового кадра.

Теперь, отсчитывая смещения от значения регистра EBP вниз, например `[EBP+8]`, получаем доступ к фактическим параметрам, а отсчитывая смещение вверх – доступ к сохраненным значениям регистров и локальным переменным. Например, выражение `[EBP-4]` определяет адрес локальной переменной, которая в программе на Ассемблере, повторяя программу на Паскале, названа именем ❷S (см. рис. 6.5). Теперь понятно, почему регистр EBP называют *базой* (base pointer) стекового кадра.

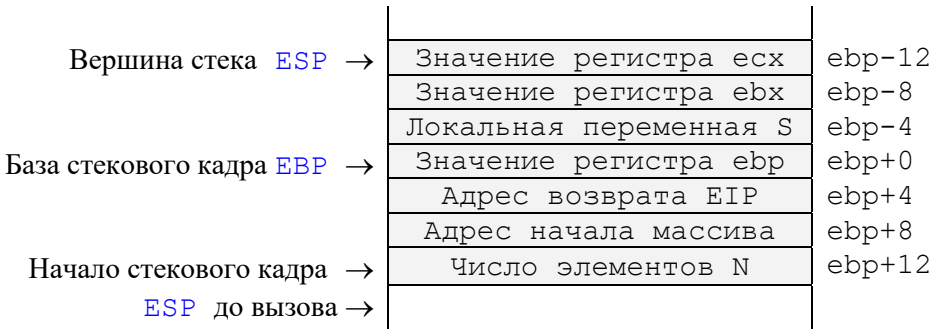


Рис. 6.5. Вид полного стекового кадра (справа показаны смещения двойных слов кадра относительно значения регистра EBP).

Итак, при входе в процедуру сначала в стеке запоминается регистр EBP и на него устанавливается база стекового кадра, затем порождаются необходимые локальные переменные, а потом спасаются используемые регистры. Эти действия носят название пролога подпрограммы (prologue). После пролога стековый кадр полностью сформирован.

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)

```
❷ S equ dword ptr [ebp-4]
```

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

```
❸ sub esp,4;  порождение локальной переменной  S
```

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

```
push eax;  порождение локальной переменной ?
```

Так часто делают компиляторы с языков высокого уровня. Можно породить локальную переменную и такой экзотической командой (чтобы запутать читателя 🙂)

```
add esp,-4;  порождение локальной переменной!
```

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

```
mov esp,ebp; уничтожение (сразу всех !) локальных переменных
```

уничтожается локальная переменная `S` (т.е. она удаляется из стека, т.к. в регистре `EBP` храниться старое значение регистра `ESP`, каким оно было до порождения локальных переменных). Переменную `S` можно было бы уничтожить и командой

```
add esp,4; уничтожение локальной переменной
```

Эта команда, однако, длиннее и более медленная. Далее восстанавливается регистр `EBP` (заметьте, база стекового кадра далее в нашей функции не понадобится). 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 подпрограммы (epilogue). После эпилога стековый кадр полностью уничтожен.

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 Free Pascal эта функция имеет следующий вид:

```
function Fact(N: ❶ byte): Longword;
begin {$R-}
  if N<=1 then Fact:=1
  else Fact:=N*Fact(N-1)
end;
```

Будем надеяться, что Вы все хорошо понимаете, как работает эта рекурсивная функция 🙂. Заметим, что в качестве параметра выбран «маленький» беззнаковый тип `byte`, факториал от бóльших значений просто не поместиться в самый большой 32-битный целочисленный тип `Longword`. Реализуем теперь этот алгоритм в виде функции на Ассемблере. Сначала заметим, что условный оператор Паскаля **`if then else`** для программирования на Ассемблере неудобен, так как содержит две ветви (**`then`** и **`else`**). Их 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)[1]
```

Здесь присутствует оператор присваивания `Fact:=1`, который для большинства вызовов функции не нужен, так как `N>1`, однако отсутствует ветвь **`else`** условного оператора. Теперь приступим к написанию функции `Fact` на Ассемблере. Сначала решим, что делать, если `Fact(N)` слишком велико, и не может быть представлено в типе `Longword` ? Выдавать при этом неправильный ответ как то «неправильно» 🙂. Для решения этой проблемы давайте сделаем так, что наша функция будет возвращать результат `-1=MaxLongword` для случая слишком большого значения факториала. Можно предложить такую реализацию этой функцию:

```
Fact proc; стандартные соглашения о связях
   push ebp;       спасаем регистр ebp
   mov  ebp,esp;   база стекового кадра
; Локальных переменных нет
   push edx;       спасаем регистр edx
```

[1] Любопытно, что многие компиляторы, использую оптимизацию под названием «устранение хвостовой рекурсии» (tail recursion elimination), могут здесь вообще заменить рекурсию на цикл!

```
; Дадим имя формальному параметру N
N   equ  dword ptr [ebp+8]; параметр N
    mov  eax,1; Fact:=1 при N<=1
    cmp  N,1
    jbe  Vozv; сейчас Fact=EAX=1
    mov  eax,N
    dec  eax
    push eax;  параметр N-1
    call Fact; рекурсия
; При возврате eax=Fact(N-1)
    cmp  eax,-1
    je   Vozv; уже большой Fact(N)
    mul  N
;<edx:eax>:=eax*N = Fact(N-1)*N
;CF=1 при edx<>0
    jnc Vozv;    пока хороший Fact(N)
    mov eax,-1; теперь большой Fact(N)
Vozv:
    pop  edx;  восстанавливаем регистр edx
    pop  ebp;  восстанавливаем регистр ebp
    ret  4
Fact endp
```

В качестве примера рассмотрим вызов этой функции оператором Паскаля `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.

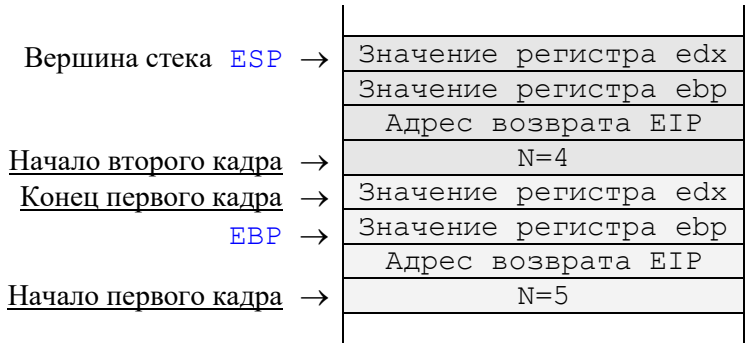


Рис. 6.6. Два стековых кадра функции `Fact`.

В качестве ещё одного примера рассмотрим реализацию с помощью *процедуры* следующей задачи. Задана константа `N=3000000` и массив X из N знаковых целых чисел, размером в двойное слово (**dd**). Надо найти количество элементов, которые больше *последнего* элемента этого массива. На языке Free Pascal эту процедуру можно записать, например, следующим образом:

```
const N=3000000
type  Mas=array[1..N] of Longint;
var X: Mas; K: Longword;
procedure NumGtLast(var X: Mas; N: Longword; var K: Longword);
{stdcall; external; Потом напишем эту функцию на Ассемблере}
var i: Longword;
begin
  K:=0;
```

```
  for i:=1 to N-1 do
    if X[i]>X[N] then inc(K)
end;
begin { Ввод X } P(X,N,K); writeln(K) end.
```

Надеюсь, Вы разберётесь, где имя K локальное, а где глобальное (нигде!). Перед реализацией процедуры NumGtLast напишем сначала фрагмент на Ассемблере для вызова этой процедуры по соглашению о связях **stdcall**:

```
sub   esp,3*4;  Место в стеке под три параметра
mov   [esp],offset X;        Адрес массива X (1-й параметр)
mov   dword ptr[esp+4],N;    Длина массива (2-й параметр)
mov   [esp+8], offset K;     Адрес  K (3-й параметр)
call  NumGtLast
```

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

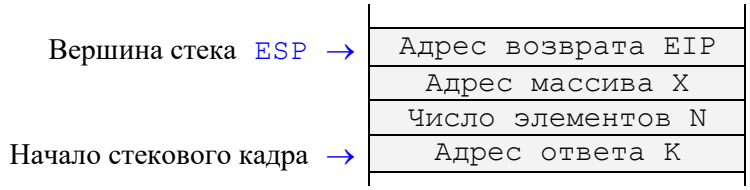


Рис. 6.7. Стековый кадр при входе в процедуру NumGtLast.

Теперь опишем саму процедуру на Ассемблере:

```
NumGtLast proc
; ПРОЛОГ процедуры:
   push ebp
   mov  ebp,esp; база стекового кадра
; сохранение регистров ❶
   push eax
   push ebx
   push ecx
   push edi
; конец ПРОЛОГА процедуры
; локальная переменная K на регистре ebx
   sub  ebx,ebx; Временная K=ebx:=0
   mov  ecx,[ebp+12];        длина массивов N
   mov  edi,[ebp+8];         адрес массива X
   jecxz Kon; для случая N=0
   dec  ecx; N:=N-1, последний элемент не смотрим
   jecxz Kon; для случая N=1
   mov  eax,[edi+4*ecx-4]; это eax:=X[N]
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
Kon:
; послать ответ из ebx в K
   mov  edi,[ebp+16]; адрес K
   mov  [edi],ebx;     послать ответ в K
; ЭПИЛОГ процедуры
; восстановление регистров ❷
   pop  edi
   pop  ecx
   pop  ebx
   pop  eax
```

```
    pop   ebp
    ret   3*4; Возврат с очисткой стека от 3-х параметров
NumGtLast endp
```

Современные компиляторы с языков высокого уровня для сохранения в стеке используемых регистров применяют другой способ:

```
; сохранение регистров ❶
  sub esp,4*4; память для регистров в стеке
  mov [ebp-4],eax;   вместо push eax
  mov [ebp-8],ebx;   вместо push ebx
  mov [ebp-12],ecx;  вместо push ecx
  mov [ebp-16],edi;  вместо push edi
```

Можно сказать, что обычно компиляторы отдают предпочтение повышению скорости работы вместо уменьшения размера программы. Как и для сохранения регистров, для их восстановления современные компиляторы применяют другой способ:

```
; восстановление регистров ❷
  mov edi,[ebp-16]; вместо pop edi
  mov ecx,[ebp-12]; вместо pop ecx
  mov ebx,[ebp-8];  вместо pop ebx
  mov eax,[ebp-4];  вместо pop eax
  mov ebp,[ebp];    вместо pop ebp
  add esp,5*4       очистка стека
```

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

```
mov edi,[ebp+16]; адрес K: edx:=(ebp+16)↑
mov [edi],ebx;    послать ответ в K: K:=(ebp+16)↑↑
```

учащиеся используют неправильную запись ответа в переменную K в виде

```
mov [ebp+16],ebx; это (адрес K)=(ebp+16)↑:=ebx
```

Другими словами, ответ пишется не в область данных, а в область стека, где хранится адрес этого ответа. Кстати, этот адрес будет уничтожен при возврате из процедуры. Некоторые учащиеся, понимая это, пытаются записать ответ в виде

```
mov [[ebp+16]],ebx; (ebp+16)↑↑:=ebx
```

К сожалению, команды такой адресации двойной косвенности в языке нашей машины нет.

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. Отметим также, что современные компиляторы могут транслировать программы с разными уровнями оптимизации, получая на старших уровнях оптимизации весьма компактный код. [v] [см. сноску в конце главы]

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 в виде так называемых *спецификаций*.

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

---

[1] В язык машины введены специальные команды: **enter** и **leave** для создания пролога и эпилога. Так, команда **enter** выполняется как две команды `mov esp,ebp` и `pop ebp`, т.е. как начало пролога. Эти команды предоставляют мало возможностей и медленно работают, ими пользуются редко.

```
var X: Longint; Y: Word;
procedure P(var Par1: Longint; Par2: Word);
var Loc1: Longword;
    Loc2: array[1..4] of word;
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,12
; 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-12]
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 add esp,12
   pop  ebp
   ret  8; и очистка 2-х параметров
P  endp
```

Перепишем теперь описание процедуры P, задав для компилятора Ассемблера дополнительную информацию в описании процедуры (спецификации заданы служебными словами **uses** и **local**):

```
P  proc uses eax ebx, Par1: dword, Par2: word
   local Loc1: dword, Loc2[4]: word [1]
comment * Начало большого комментария
;  Ассемблер автоматически вставляет сюда команды:
   push ebp
   mov  ebp,esp; база стекового кадра
;  порождение переменных Loc1 и Loc2
   sub  esp,12; порождение Loc1 и Loc2
;  сохранение регистров
```

[1] В заголовке процедуры перед **uses** и **local** можно дополнительно указывать соглашение о связях (**C**, **stdcall** или **Pascal**), в этом случае они перекрывают соглашения, указанные в директиве **.model** в начале программы. Кроме того, там же можно задать видимость имени процедуры с помощью ключей **public**, **export** (работает как **extern**) или **private**, по умолчанию **public**, например:

```
P proc stdcall private uses eax ebx, Par1: dword
```

```
   push eax
   push ebx
;  Ассемблер автоматически вставляет директивы
Loc1 equ dword ptr [ebp-4]
Loc2 equ dword ptr [ebp-12]
Par1 equ dword ptr [ebp+8]
Par2 equ word ptr  [ebp+12]
;  это локальные имена, не видимые извне процедуры !
*  Конец большого комментария
       Тело процедуры
;  Вместо каждой ret Ассемблер вставляет команды:
;  восстановление регистров
   pop  ebx
   pop  eax
;  уничтожение Loc1 и Loc2
   mov  esp,ebp
   pop  ebp
   ret  8; очистка 2-х параметров
;  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: dword, oc2[4]: word
       Тело процедуры
   ret
P  endp
```

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, имеющую служебное имя **invoke**:

```
invoke P,Addr 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. Необходимо также учесть, что все параметры записываются в стек в формате **dword**, в частности, короткие регистры (беззнаково) расширяются до 32-бит.

Макрокоманда **invoke** сама вставляет в процедуру команд для пролога и эпилога посредством вызова *стандартных* макрокоманд с именами **PrologueDef** и **EpilogueDef**. Можно запретить вызывать эти вызовы 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).

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. Как известно, 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
```

В прототипе указано имя процедуры и размер всех её параметров. Если процедура в **proto** 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.

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. С другой стороны, X может быть локальной переменной в стеке, например, **Addr** [EBP-8], что для операторов **offset** X невозможно.

Студенты, изучающие данный курс, должны уметь сами записывать все необходимые команды для реализации вызова, пролога и эпилога процедур и функций.[1]

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 использование 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, при разработке архитектуры ЭВМ этому уделяется большое внимание. [vi] [см. сноску в конце главы]

6.13. Дальние процедуры на Ассемблере

Шарлатан тот, кто не может объяснить восьмилетнему ребёнку, чем он занимается.

Курт Воннегут. «Колыбель для кошки»

Этот раздел только для прилежных и продвинутых учащихся 🙂.

Наряду с близким, в языке машины есть также *дальний* (межсегментный) вызов процедуры **call** op1, с параметром форматов m48=m16:m32 (это шестибайтный формат **df** – Define Far ptr) или i48=i16:i32, такие операнды обычно обозначается как seg:offset.

Для понимания работы дальних вызовов процедур предварительно нужно изучить так называемые *уровни привилегий*. Это очень сложная тема, однако можно отметить, что в рамках данного курса команды дальнего вызова процедур в программировании на Ассемблере не используются, с ними работают только служебные программы для вызова системных процедур Windows (да и то очень редко, сейчас для этого используются другие методы). Так что это теоретическая тема, необходимая, однако, для *глубокого* понимания архитектуры изучаемого компьютера.

6.13.1. Уровни привилегий

В этом мире люди ценят не права, а привилегии.

Генри Луис Менкен, сатирик

- Уровни привилегий призваны обеспечить безопасность функционирования вычислительной системы, они определяют возможность для программы 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 предумышленного 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.

---

[1] В 64-битном режиме программисту на Ассемблере при написании и вызовах процедур *настоятельно рекомендуется* использовать только встроенные средства (встроенные макрокоманды) Ассемблера. В то же время *понимать*, как это выглядит на языке машины, является *совершенно необходимым* для уровня университетского образования.

Уровень привилегий определяется режимом работы процессора в защищённом режиме, соответственно различают привилегированный (supervisor mode) и непривилегированный (user mode) режимы работы. Для текущей программы режим работы процессора 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 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 (архитекторы здесь перестарались…).

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 дескриптор различных объектов (сегментов, шлюзов вызова, задач, локальной дескрипторной таблицы 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 храниться 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)<=DPL, значение Max(CPL,RPL) называется эффективным уровнем привилегий (соображаем, что CPL находится в селекторе регистре CS, RPL в селекторах регистров DS или ES, а 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 свой (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=DS(RPL)=3, а у дескриптора кодового сегмента DPL=0, это значит, что сейчас программа пользователя выполняет процедуру из кодового сегмента операционной системы, предоставив ей для работы свой сегмент данных. Когда 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. [vii] [см. сноску в конце главы]

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 из него на любом этаже, ведь этим лифтом пользуются и простые служащие и начальники 😕).

---

[1] Заметим, однако, что возможность выполнения привилегированных 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.

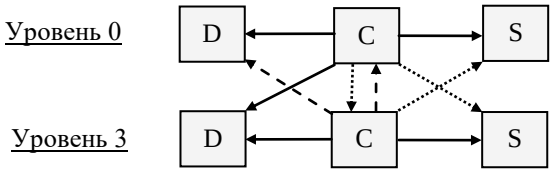


Рис 6.8. Доступ из кодового сегмента к другим сегментам с разным уровнем привилегий (сегменты: D –данных, C – кода, S – стека). Линии: сплошная – разрешено, точками – не разрешено, пунктир – только к согласованному (подчинённому) сегменту.

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

`call` `op1` = **`call`** `seg:offset`

и выполняется по следующей схеме:

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

Способ перехода на вызываемую процедуру определяется полем `seg` в команде вызова, это поле 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=seg:offset`.

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` (такие 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 (подчиняемым) или *согласованным*, процедуру из такого сегмента разрешается вызывать всем, однако выполняться она будет с уровнем привилегий вызывающего сегмента, т.е. `CPL>=DPL` и `CPL` не меняется. Получается, что у подчиняемого сегмента нет своего собственного уровня привилегий. То есть в квартиру гостя пустили, но делать там всё, что можно хозяевам (например, выполнять привилегированные команды), 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. Главное назначение подчиняемых сегментов – создать программный код, разделяемый многими вычислительными процессами, работающими на разных уровнях привилегий (обычно это общие библиотеки).

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*. Получается, что вызов через шлюз задаёт не прямой, а косвенный вызов системной процедуры. 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 эти шлюзы не должны вести на менее привилегированный

уровень (CPL>=DPL) (можно пользоваться только разрешёнными лифтами, но все они должны ехать 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 сам *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 записывает в системный стек вызываемой процедуры (вслед за регистрами `EIP` и `CS`). Это от 0 до 31 фактических параметров системной процедуры, которые записала 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

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)` **`mod`** 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. поле `Count=2`) при вызове процедуры через шлюз. Индекс `P` у регистров обозначает вызывающую программу пользователя, а индекс `S` – системную процедуру.

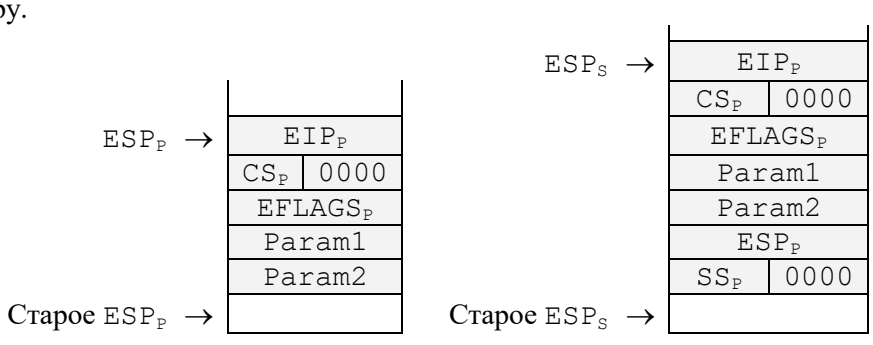


Рис. 6.9. Вид стека программы и системного стека перед командой возврата **`retf`** `8` (два параметра, `Count=2`) при вызове процедуры через шлюз.

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

`ESP:=(ESP + Longword(i16))` **`mod`** 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 после возврата из процедуры не заметит, что работа велась не в её стеке, а в системном.

На этом заканчивается изучение процедур в языке Ассемблера.

6.14. Соглашение о связях в 64-битном Ассемблере

Это ещё не конец. Это даже не начало конца. Но это, возможно, конец начала.
Уинстон Черчилль, 1942 год,
это Антиметáбола

Этот раздел только для прилежных и продвинутых учащихся.

В 64-битном режиме работы процессоров Intel в каждой операционной системе у всех языков осталось только одно соглашение о связях. Например, для ОС Windows это соглашение разработано на основе Intel ABI (application binary interface) и называется **vectorcall** (так как для передачи параметров, в частности, используются векторные XMM-регистры).

По этому соглашению параметры, длиннее 64-х бит (кроме XMM регистров), передаются только по ссылке. При передаче по значению все параметры должны иметь длину 64 бита (но не гарантируется, что для *маленьких* параметров производится их знаковое или беззнаковое расширение). Таким образом, под каждый параметр всегда отводится 8 байт.

Не надо путать, в языках программирования высокого уровня фактические параметры при передаче в процедуру могут преобразовываться. Например, для языка Free Pascal:

```
var A: byte;
procedure P(X: qword);
P(A);
```

Здесь при передачи параметра A в процедуру его значение будет расширяться 7-ю *нулевыми* байтами.

Первые четыре (слева направо) целочисленных и вещественных параметров передаются на регистрах, целые на RCX, RDX, R8 и R9, а вещественные в младших частях регистров XMM0-XMM3 (старшие части можно не обнулять) и дублируются на соответсвующие целочисленные регистры.[1]. Остальные параметры (длиной по 8 байт) передаются через стек (и уже в обратном порядке).

Заметим, что при таком способе передачи, все фактические параметры подпрограммы располагаются в стеке естественно и «красиво». Они находятся сразу вслед за адресом возврата, слева направо по возрастанию их номеров в списке параметров и сверху вниз по стеку. При этом i-й параметр (1≤i≤N) находится по адресу `[RSP+8*(i-1)]`.

Отметим, что, даже если параметры передаются только на регистрах, в стеке, тем не менее, перед вызовом требуется зарезервировать место для 4-х параметров длиной 8*4=32 байта (эта область называется register parameters area или home location или shadow space), как и при передаче параметров только через стек. Это место по-русски принято называть теневой областью (shadow storage) стекового кадра, она резервируется, даже если у подпрограммы меньше 4-х параметров (или их нет вообще). Предполагается, что, при необходимости, подпрограмма может сохранить на это место старые значения данных регистров (командами **mov**, а не **push**), чтобы использовать их до того, как из них прочитаны входные значения параметров (хотя значения регистров RCX, RDX, R8 и R9 можно и не восстанавливать перед возвратом). По существу, это просто (рабочие) локальные переменные подпрограммы, например, в них можно сохранять постоянные (nonvolatile) регистры (см. далее). Но главное, что такой способ упрощает определение *места расположения* каждого фактического па-

[1] Вообще говоря, такое дублирование параметров по стандарту обязательно только для подпрограмм с переменным числом параметров, например, для таких, как write или printf, но можно делать его всегда.

раметра в стеке. Также нужно, чтобы перед возвратом был очищен стек вещественных регистров FPU (это одна команда **emms**) и очищен флаг направления DF.

В том случае, если какие-то из первых 4-х параметров имеют длину 16 байт (это типы **oword** и **xmmword**), то они помещаются только на соответствующий регистр XMM0-XMM3, а целочисленные регистры для них не используются (имеют неопределённые значения, но место для них есть в стеке!). Отсюда делаем вывод, что иметь в подпрограмме более четырёх параметров нехорошо 🐷).

Теневая область не резервируется, если у подпрограммы задать модификатор NoStackFrame, в этом случае параметры передаются только на регистрах (и параметров не более 4-х). Это уже упоминаемые в разделе 6.10.1 (о)конечные подпрограммы (leaf function).

Целое значение возвращается функцией на регистрах AL, AX, EAX, RAX и <RDX:RAX>, а вещественное длиной до 16 байт – на регистре XMM0 (или в его младшей части). Отметим, что от *теневой области* стекового кадра стек очищает вызывающая программа (как и в соглашении **cdecl** языка C). Заметим также, что для доступа к стековому кадру чаще всего используется не регистр RBP а непосредственно указатель вершины стека RSP. Для программиста на Ассемблере это и не совсем удобно, так как приходится пересчитывать смещения при каждом изменении размера области локальных переменных процедуры (следовательно, надо сначала полностью сформировать стековый кадр).

Регистры RBX, RBP, RDI, RSI, RSP, R12-R15 называются *постоянными* (immutable, nonvolatile) или *неизменяемыми* (nonvolatile – часто встречается плохой и непонятный перевод «энергонезависимые»), т.е. вызываемая подпрограмма должна сохранять их перед изменением и восстанавливать перед возвратом.[viii] [см. сноску в конце главы]. Наоборот, регистры RAX, RCX, RDX, R8-R11 считаются непостоянными (volatile, «энергозависимыми»), т.е. их не надо сохранять перед использованием; таким образом, регистры со значениями параметров RCX, RDX, R8 и R9 можно не восстанавливать перед возвратом. Аналогично вещественные регистры XMM0-XMM5 считаются непостоянными, а XMM6-XMM15 – постоянными. Отметим, что регистры XMM0-XMM15 являются младшими частями 256-разрядных регистров YMM0-YMM15 и «совсем младшими» частями 512-разрядных регистров ZMM0-ZMM15.

Кроме того, все подпрограммы могут использовать для своей работы область перед вершиной стека *без сдвига* самой вершины. Эта область в 64-битном режиме Windows гарантированно имеет длину не менее 128 байт и называется *красной зоной* (red zone) стека. Красная зона удобна для размещения локальных переменных, так как для доступа к ним достаточно короткого (однобайтного) смещения от указателя вершины стека RSP.

Ясно, что красная зона это «совсем временное» хранилище данных, оно существует, пока наша подпрограмма не вызовет другую подпрограмму. Для сохранения этой зоны подпрограмма, перед вызовом другой подпрограммы, должна увеличить свой стековый кадр, сдвинув вершину стека (например, командой `sub rsp,128`), а перед возвратом выполнить команду `add rsp,128`.

Далее требуется, чтобы граница стекового кадра была выровнена на границу 16 байт (16-byte-aligned) после входа в каждую подпрограмму. Одна из причин этого заключается в том, что многие векторные машинные команды производят обмен XMM регистров с памятью только при выравнивании данных в памяти по границе 16 байт. Для обеспечения такого выравнивания есть два способа.

Первый способ применяется, когда главная программа знает как «устроена» вызываемая подпрограмма, сколько у неё локальных переменных, сколько она спасает и восстанавливает регистров и т.д. В этом случае главная программа может целиком взять на себя построение стекового кадра. Например, пусть есть процедура P (для 64-битных ОС):

```
procedure P(var X1: byte; X2: word; X3: longint; X4: double;
  var X5: int64; X6: word); vectorcall; external;
  var A1: word; A2: array[1..5] of byte;
begin { Используются регистры rbx,rsi } ... end;
```

Тогда её вызов будет, например, выглядеть так:

```
; Пусть сейчас RSP кратен 16
; ↓ ↓ В стеке надо отвести для параметров
; 4*8{X1..X4}+2*8{X5..X6}=48 байт
```

```
; ⬇ ⬇ Для локальных переменных и регистров
; 7{A1+A2}+16{rbx,rsi}=23 байта
; ⬇ ⬇ +1=72, чтобы RSP был кратен 8, но не 16
   sub     rsp,72;          стековый кадр
   mov     rcx,offset X1;   1-й параметр
   movzx   rdx,X2;          2-й параметр, rdx:=qword(X2)
   movsx   r8,X3;           3-й параметр, r8:=int64(X3)
   mov     r9,X4;           4-й параметр, r9:=X4
; ⬇ ⬇ Копия 4-го параметра, xmm3[0..63]:=X4
   movhps  xmm3,X4
   movzx   rax,X6;          можно movzx eax,X6
   mov     [rsp+40],rax;    6-й параметр, qword(X6)
   mov     rax,offset X5
   mov     [esp+32],rax;    5-й параметр, offset X5
   call    P;               после call значение RSP кратно 16!
   add     rsp,72;          очистка стека
```

Процедура P будет выглядеть так:

```
P  proc
; Спасение регистров rbx,rsi
   mov [rsp+48],rbx;      спасение RBX
   mov [rsp+56],rsi;      спасение RSI
; ⬇ ⬇ Локальные переменные A1 и A2
LocA1 equ word ptr [rsp+64]
LocA2 equ byte ptr [rsp+66]
; Тело процедуры
; Восстановление регистров rbx,rsi
mov rbx,[rsp+48]
mov rsi,[rsp+56]
ret
P  endp
```

Пример простой функции суммирование массива:

```
N equ 100000
.data
; type Mas: array[1..N] of longint;
A dd N dup (?); var A: Mas;
.code
; function Sum(var X: Mas; N: longword): int64;
Sum proc; rcx=offset X, rdx=N
    sub   eax,eax; rax:=0 ⚠
; ⬇ ⬇ for rdx:=N downto 1 do
; ⬇ ⬇     rax:=rax+X[rdx]
@@: movsx r8,[rcx+4*rdx-4]
    add   rax,r8
    sub   rdx,1
    jnz   @B
    ret
Sum endp
Start: ; ввод A
    sub rsp,40; для параметров-регистров+8
    mov rcx,offset A; 1-й параметр
    mov rdx,N;        2-й параметр
; ⬇ ⬇ после call RSP кратен 16!
    call Sum
```

```
    add rsp,40; очистка стека
; ⬇ ⬇ сейчас RSP снова кратен 16!
    outintln rax,,"Сумма="
exit
end Start
```

Второй способ выравнивания стека применяется, когда Вы вызываете из Ассемблера «чужую» подпрограмму, например, с переменным числом параметров, неизвестным объёмом локальных переменных, неизвестным числом используемых регистров и т.д. В этот случае стандартные соглашения о связях требуют, чтобы значение RSP было кратно 16 непосредственно перед выполнением команды **call**, а полное построение выровненного стекового кадра является уже ответственностью самой подпрограммы. Для гарантии после полного формирования стекового кадра в подпрограмме можно поставить команду выравнивания стека `and rsp,0FFFFFFFFFFFF0h`, программисты на Ассемблере, пишут её как `and rsp,-16` (Вы же понимаете, что это одно и тоже 🙂).

Для листовых подпрограмм (которые сами никого не вызывают), имеющих не более 4-х параметров, стековый кадр можно фактически не использовать. В последнем примере из подпрограммы можно удалить строки, помеченные серым фоном.

Для операционных систем семейства Linux используется немного другое соглашение о связях с официальным названием System V AMD64 ABI. По этому соглашению уже первые шесть целочисленных параметра передаются на регистрах RDI, RSI, RCX, RDX, R8 и R9, остальные параметры передаются через стек (в обратном порядке). Первые 8 вещественных параметров передаются на 128-битных регистрах XMM0-XMM7 (типы Single и Double – в младших частях этих регистров). Дополнительные вещественные параметры передаются через стек в обратном порядке (команда **push** здесь уже не подходит). Регистр RAX при необходимости используется как счётчик переменного числа параметров, а R10 как ссылка на стековый кадр предыдущей подпрограммы в цепочке вызовов (static chain pointer). Регистры RBX, RBP, R12-R15 считаются постоянными, если они используются в подпрограмме, то их надо запомнить и потом восстановить. Как видим, всё сложно 😖.

Заметим, что при таком способе передачи параметров, в ОС Linux, в отличие от Windows, процедура не сможет узнать «настоящий» порядок параметров, например, если поменять местами первый целый и первый вещественный параметры, то вызов на языке машины не изменится! 😟

Список литературы по 64-битному Ассемблеру см. в Главе 17.

Вопросы и упражнения

Усердие всёпревозмогает!
Козьма Прутков

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. Дана команда `cmp eax,X`. Написать эквивалентный фрагмент на Ассемблере, но без использование этой команды.
5. Чем понятие процедуры в Ассемблере отличается от понятия процедуры в Паскале ?
6. Обоснуйте необходимость принятия стандартных соглашений о связях между процедурой и вызывающей её программой.
7. Что такое прямой и обратный порядок передачи параметров в процедуру? Когда может возникнуть необходимость в обратном порядке передачи параметров в процедуру через стек ?
8. Что такое стековый кадр ?
9. Обоснуйте, какие из пунктов стандартных соглашений о связях **необходимы** для обеспечения *рекурсивных* вызовов процедур и функций. *рекурсивных* вызовов процедур и функций.
10. Почему для соглашения **stdcall** необходимо, чтобы при возврате из процедуры стек находился точно в том же состоянии, что и перед началом вызова этой процедуры ?
11. Какие метки в Ассемблере записываются с двумя двоеточиями L:: ?

[i] Для продвинутых читателей. Рассмотрим, как работает со стеком ОС Windows. 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 аварийная ситуация (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_VIOLATON), 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 счете задачи выделенная под стек память может увеличиваться в пределах отведённого максимального выделенного размера памяти (обычно 1 Мб, но при создании выполняемого файла может задаваться и другой размер стека).

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 этой ошибки. Вот для нормальной работы этой процедуры и предназначен этот «неприкосновенный запас» в виде двух последних страниц стека. И, наконец, перед последней «красной» сторожевой страницей (RED_PAGE_GUARD), как и после стека, обязательно располагается «чужая» страница, попытка доступа в которую, как уже говорилось, вызывает исключение (ошибку) EXCEPTION_ACCESS_VIOLATION.

Резервирование памяти в стеке обычно производится командой `sub esp,n`, для больших `n` (при резервировании памяти, больше, чем размер страницы) это может вызвать ошибку, если будет попытка «перескочить» сторожевую страницу. В библиотеке операционной системы есть специальная подпрограмма (в библиотеке языка C она имеет имя `__chkstk`), которая для параметра n «осторожно, мелкими шагами» движется по стеку, резервирую память кусочками, не превышающими размером страницу (тише едишь, дальше будешь ☺).

В Ассемблере обычно в начале работы под стек отводится одна страница директивой `.stack 4096` (байт). Таким образом, по умолчанию сначала под стек отведены всего три страницы, одна «настоящая», одна сторожевая и одна резервная (последняя в стеке, ближе к началу памяти). Впрочем, некоторые операционные системы изначально делают стек пустым (не выделяют ни одной «настоящей» страницы). Заметим также, что после роста, снова уменьшать в размерах стек ОС Windows не может (впрочем, страницы стека в режиме виртуальной памяти, как обычно, «свопируются» в файл подкачки). Изложенный выше механизм работы задачи со стеком существенно усложняется, если задача запускает несколько вычислительных потоком (или нитей – threads). Все потоки одной 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 и в курсе по операционным системам.

Забегая вперёд отметим, что, в отличие от стека задачи, под так называемый *системный* стек, используемый функциями из библиотек Windows, отводится всего три страницы (четыре в 64-битном режиме), т.е. он совсем маленький и динамически расти не может (системные процедуры экономно и качесвенно работают со стеком).

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.

[ii] Для продвинутых читателей. Эти стековые операции позволяют «напрямую» читать и изменять значения флагов. Однако, при работе обычного пользователя в защищённом режиме некоторые биты регистра флагов (например, флаг запрета прерываний `IF` и двухбитовое поле `IOPL`, задающее уровень привилегий ввода/вывода) при чтении из стека в регистр `EFLAGS` *не меняются*. Кроме того, обмениваться значениями флагов (кроме `OF`) с регистром `EFLAGS` можно и с помощью команд

`lahf` ; `AH:=0000 0010b` **`or`** `EFLAGS[7,6,4,2,0]`

и

`sahf` ; `EFLAGS[7,6,4,2,0]:=AH[7,6,4,2,0]`

Эти команды предназначены для одновременного изменения сразу нескольких флагов. Видно, что таким образом нельзя изменить флаги `EFLAGS[5,3,1]`, они зарезервированы фирмой Intel.

Любопытно, что пары команд обмена значениями между регистром флагов и регистрами общего назначения, например **`pushfd`** и **`pop`** `eax` вообще ничего не пишут в память, использую хитрый механизм, по которому посылаемое в память значение сначала записывается на особые регистры «очередь записи», откуда сразу же и беруться следующей командой, таким образом вообще не попадая в кэш память.

[iii] Для продвинутых читателей. Различают *близкий* (внутрисегментный) и *дальний* (межсегментный) вызовы процедуры. Для близкого вызова параметр `op1` может иметь следующие форматы: `i16`, `r16`, `m16`, `i32`, `r32` и `m32`. Как видно, по сравнению с командой обычного безусловного перехода **`jmp op1`** здесь не реализован близкий короткий относительный переход **`call i8`**, он бесполезен, 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`.

Аналогично, у команды возврата 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 «хитрой» 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

[iv] Для продвинутых читателей. В стеке, в частности, порождают и уничтожают локальные переменные (в смысле языков высокого уровня) с помощью команд **push** и **pop**, или изменения значения регистра `ESP`. Важно понять, что в 32-битном режиме, например, команда **pop** на самом деле уничтожает локальную переменную, ее нельзя «восстановить» командой сдвига указателя вершины стека `ESP`. Например:

```
S equ dword ptp [esp]; S на вершине стека
  sub esp,4; порождение локальной переменной S=?
  mov S,1;   S:=1
  add esp,4; уничтожение локальной переменной S
  sub esp,4; порождение новой S с неопределённым значением!
```

Нельзя гарантировать, что новая переменная будет иметь прежнее значение `S=1`. Здесь всё дело в том, что между двумя последними командами **`add esp,4`** и **`sub esp,4`** могло произойти так называемое непривилегированное прерывание (о прерываниях будет говорится в другой главе), при этом аппаратура компьютера сама пишет в наш стек свои данные, «затирая» переменную `S`. Перед возвратом на команду **`sub esp,4`** стек будет очищен, но старое значение `S` уже испортится ⚠.

Стоит отметить, что в 64-битном режиме это уже не так: все прерывания там используют свой собственный *системный* стек, а стек пользователя превращается просто в ещё одну область памяти, которую уже нельзя в полной мере назвать стеком ☹. Исходя из этого те «оконечные» функции (leaf function), которые сами никого не вызывают, часто используют для своих локальных переменных область перед вершиной стека *без сдвига* самой вершины. Эта область в 64-битном режиме Windows гарантированно имеет длину не менее 128 байт и называется *красной зоной* (red zone) стека. Красная зона удобна для размещения локальных переменных, так как для доступа к ним достаточно короткого (однобайтного) смещения от указателя вершины стека `RSP`.

[v] Для продвинутых читателей. В качестве примера рассмотрим компиляцию функции:

```
function F(X: Longint):Longint;
  var Y: Longint;
begin {$R-} Y:=3*X+1; F:=5*Y-7 end;
```

на разных уровнях оптимизации компилятора с языка Free Pascal: на нулевом (без оптимизации, «один в один»), этот уровень задаётся ключом компилятора `-O0`, и на третьем уровне оптимизации (ключ `-O3`):

```
F proc
  push ebp
  mov  ebp,esp
  sub  esp,4;    для var Y
  push ebx
  mov  eax,[ebp+8]; X
  mov  ebx,3
  imul ebx;      eax:=3*X
  inc  eax;      eax:=3*X+1
  mov  [ebp-4],eax; Y:=3*X+1
  mov  ebx,5
  imul ebx;      eax:=(Y=eax)*5
  sub  eax,7;    F:=eax:=5*Y-7
  pop  ebx
  mov  esp,ebp; уничтожение Y
  pop  ebp
  ret  4
F endp
```

```
F proc
; F:=5*(3*X+1)-7=15*X-2
  mov eax,[esp+4];        X
  lea eax,[eax+4*eax];    5*X
  lea eax,[eax+2*eax-2]; 15*X-2
  ret 4
F endp
```

Как говорится, почувствуйте разницу. Вот так и должен писать на Ассемблере *хороший* программист 🙂. Впрочем, такой результат получается только в режиме работы компилятора Free Pascal с директивой `{$R-}`, т.е. без контроля выхода значений за допустимые диапазоны. А вот с директивой `{$R+}` картина компиляции с оптимизацией будет другой:

```
F proc
; F:=5*(3*X+1)-7=15*X-2
  push edx
  mov  eax,15
  imul dword ptr [esp+8]; <edx:eax>:=15*X
  jo   Range_Checking_Error
  sub  eax,2;             15*X-2
  jo   Range_Checking_Error
  pop  edx
  ret  4
F endp
```

Как видим, контроль обходится дорого.

---

vi Для продвинутых читателей. Очевидно, что особенно интенсивно используется область, близкая к вершине стека. Для повышения эффективности, начиная с процессоров Pentium 4 Net Burst, реализован особый механизм RAS (Return Address Stack), при этом область, близкая к вершине стека, кэшируется в быстродействующей памяти (фактически на регистрах).

Следует понять, что наиболее уязвимым элементом стекового кадра любой процедуры является адрес возврата в главную программу. Значительная часть вредоносных программ, при внедрении в вычислительную систему, делает это, подменяя адрес возврата на переход в нужное им место оперативной памяти. При этом используется тот факт, что адрес возврата находится в стеке среди прочих объектов (параметров и локальных переменных процедуры).

В современных процессорах делаются попытки защитить в стеке адреса возврата из процедур. Для этих целей фирма Intel разработала технологию CET (Control-flow Enforcement Technology). Основных элементом этой технологии является так называемый *теневой стек* (shadow stack), который ведётся самим процессором. На вершину теневого стека указывает служебный регистр `SSP` (Shadow Stack Pointer). В теневом стеке автоматически сохраняется информация об адресах возврата для команды **`ret`**. При возврате из процедуры процессор будет проверять адрес возврата, заносимых в стек командой **`call`** (и особыми командами **`int`**), который сохранен в стеке программы, с тем, который сохранен в теневом стеке и при их несовпадении будет возникать исключительная ситуация. Кроме того, теневой стек позволяет очень эффективно реализовать предсказание переходов по команде возврата **`ret`** для конвейера.

Теневой стек помещается в отдельную защищённую область памяти, указатель на которую хранится в так называемом сегменте состояния задачи TSS, эта тема будет изучаться далее в главе, посвящённой системе прерываний. За активацию режима CET отвечает специальный флаг `CET` в управляющем регистре `CR4`. Для работы с теневым стеком реализован набор служебных регистров и новых привилегированных команд. Механизм CET реагирует на все команды вызова процедур и возврата, а также на команды прерываний.

[vii] Для продвинутых читателей. Не надо понимать высший уровень привилегий слишком буквально, в том смысле, что программе, работающей на более привилегированном уровне «всё можно». Как видно из рис. 6.8, программе, работающей на нулевом уровне привилегий, *запрещено* как передавать управление в непривилегированную программу на третьем уровне привилегий, так 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 будет значить, что привилегированная программа *осознанно* идёт на риск выполнения ненадёжного кода.

[viii] Компьютерная память называется энергонезависимой (т.е независимой от электрического питания, nonvolatile), если она сохраняет данные при отключении питания. Аналогично, энергонезависимые регистры сохраняют неизменные значения при «отключении» от главной программы, уходе в подпрограмму и последующим возврате из неё. Наоборот, непостоянные (энергозависимые – volatile) регистры могут изменить свои значения при возврате из подпрограммы.

Любопытно, что в некоторых языках (например, в C) есть класс памяти с таким же названием `volatile`. Переменные этого класса могут менять свои значения в непредсказуемые моменты времени и независимо от выполнения программы, в которой они описаны  .