Архитектура современных вычислительных устройств
Лекции 6–7. Организация памяти

Макаров П. А.

октябрь, ноябрь 2024 г.

Содержание

1 Введение

В процессорах Intel 8086 память представлялась в виде сегментов размером по 64 КБайта. Суммарный объём программно доступной памяти не превышал 1 МБайт, а архитектура IBM PC дополнительно ограничивала максимальный размер возможной оперативной памяти объёмом в 640 КБайт (так называемая “стандартная память”).

Пространство оперативной памяти предназначено для хранения и кодов инструкций программы и её данных, для доступа к которым имеется много режимов адресации. Память может логически организовываться в виде одного или множества сегментов произвольной длины. В реальном режиме этот размер фиксирован.

Для управления распределением памяти разработаны механизмы сегментации и страничной трансляции адресов, которые могут применяться как совместно, так и по отдельности.

Сегментация — это средство организации логической памяти на прикладном уровне. Страничная трансляция адресов применяется на системном уровне для управления физической памятью.

Сегменты и страницы могут выгружаться из физической оперативной памяти на жёсткий диск (в так называемый “файл подкачки” или специальный swap-раздел), и по мере необходимости подкачиваться с него обратно в физическую память. Таким образом реализуется виртуальная память.

Впервые средства поддержки виртуальной памяти появились в процессорах Intel 80286, но удобный для использования вид приняли только в 32-битных процессорах (начиная с процессора Intel 80386).

2 Адресные пространства и эффективный адрес

Различают три адресных пространства: логическое, линейное и физическое.

По сочетанию сегментации и страничной трансляции различают две основных модели памяти:

1. **Сегментная модель.** Приложение использует несколько сегментов памяти: для кода, данных, стека, и может переключаться между ними, используя селекторы сегментов. В этой модели приложение оперирует логическими адресами.

2. **Плоская модель памяти.** Приложению для всех целей выделяется единственный сегмент. В этой модели приложение оперирует

линейными адресами. Плоская модель гораздо проще и удобнее в обращении, и используется в современных операционных системах.

Логический адрес состоит из селектора сегмента и эффективного адреса, называемого также смещением (offset). Логический адрес образуется парой значений, которые записываются в форме `Seg:offset`. Селектор сегмента, хранится в старших 14 битах сегментного регистра, участвующего в адресации конкретного элемента памяти. По значению селектора из специальных таблиц дискрипторов сегментов, хранящихся в памяти, извлекается начальный адрес сегмента.

Так как каждая задача может иметь до 16 Кбит (2^{14}) селекторов, а смещение, ограниченное размером сегмента, может достигать 4 ГБайт (2^{32}, то логическое адресное пространство для каждой задачи может достигать 64 ТБайт. Это максимальное пространство виртуальной памяти, доступное программисту при использовании сегментной модели памяти. Операционная система может ограничивать число доступных сегментов и их конкретные размеры.

Эффективный адрес. При обращении к памяти для доступа к данным, как и при формировании адреса перехода, процессор строит эффективный адрес, который может включать до четырёх компонентов. Такой сложный способ задуман для облегчения доступа к элементам массива и другим сложным структурам данных.

Основными частями эффективного адреса являются базовый адрес элемента `BASE`, номер элемента `INDEX`, масштабный коэффициент `SCALE` и `DISPLACEMENT` — смещение внутри элемента. Массив может состоять из байтов, слов, двойных, учетверённых слов и т. д. Это и учитывается масштабным коэффициентом.

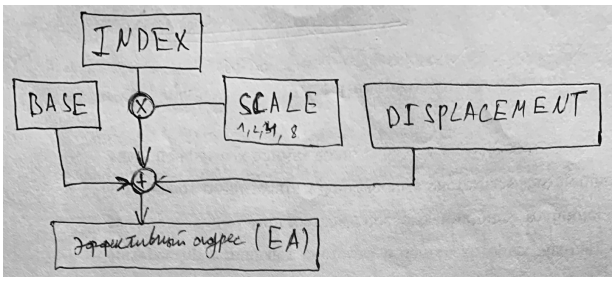

Рис. 1: Формирование эффективного адреса

Компоненты эффективного адреса могут быть константами, определяемыми в инструкции, находиться в регистрах или размещаться в ячейках памяти. Это определяется типом используемой адресации операндов.

В 64-битных процессорах появилась возможность адресации данных относительно текущего значения указателя инструкций `RIP`.

3 Распределение адресного пространства

Не следует думать, что термины **адресное пространство** и **оперативная память** эквивалентны.

Определение 1. ***Адресное пространство*** *— это набор адресов, которые умеет формировать процессор.*

Совсем не обязательно все эти адреса отвечают реально существующим ячейкам памяти. В зависимости от модификации компьютера и состава его периферийного оборудования, распределение адресного пространства может несколько различаться. Тем не менее, размещение основных компонентов системы довольно строго унифицировано.

Сегментная структура программ. Обращение к памяти почти всех процессоров семейства x86, за исключением 64-битных вариантов, работающих в длинном режиме (Long mode), осуществляется посредством сегментов — логических образований, накладываемых на любые участки физического адресного пространства.

В процессорах Intel 8086 начальный адрес сегмента, деленный на 16_{10}, т. е. без младшей шестнадцатеричной цифры, заносился в один из сегментных регистров, после чего мог быть получен доступ к участку памяти, начинающегося с заданного сегментного адреса. В последующих моделях процессоров (как в i80286, так и в моделях архитектуры IA-32) базовый адрес сегмента паняти определялся немного сложнее, но здесь по-прежнему использовались сегментные регистры.

Каким образом понятие сегментов памяти отражается на структуре программы? Структура программы определяется, с одной стороны, архитектурой процессора (если обращение к памяти возможно только с помощью сегментов, то и программа должна состоять из сегментов), а с другой — особенностями той операционной системы, под управлением которой эта программа будет выполняться. Наконец, на структуру программы влияют также и правила работы выбранного транслятора — разные трансляторы предъявляют несколько различающиеся требования к исходному тексту программы.

Описание каждого сегмента в NASM начинается с ключевого слова `segment`, сопровождаемого некоторым именем. Имена сегментов выбираются вполне произвольно, но, как правило, им дают вполне определённые имена: `code`, `data` и `stack` (см. пример ниже).

```
segment data
hello: db 'Hello, world!', 0x0D, 0x0A, '$'
```

```
segment code
start:
  mov ax, data
  mov ds, ax

print:
  mov dx, hello
  mov ah, 09
  int 0x21

quit:
  mov ax, 0x4C00
  int 0x21

segment stack class=stack
  resb 512
```

Порядок описания сегментов в программе, как правило, не имеет значения. Однако важно понимать, что в оперативную память компьютера сегменты попадут в том же порядке, в каком они описаны в программе (если специальными средствами ассемблера не задать иной порядок загрузки сегментов в память).

Сегменты вводятся в программу с помощью директивы ассемблера `segment`. К директивам ассемблера относятся не только объявления сегментов, но и ключевые слова, описывающие тип используемых данных (db, и др.), а также специальные описатели сегментов вроде stack и т. д.

Вообще, директивы служат для передачи транслятору служебной информации, которой он пользуется в процессе трансляции программы. Однако в состав выполнимой программы, состоящей из машинных кодов, эти строки не попадают, так как процессору, выполняющему программу, они не нужны. Другими словами, операторы типа segment не транслируются в машинные коды, а используются лишь самим ассемблером на этапе трансляции программы.

При загрузке программы сегменты размещаются в памяти, как показано на рисунке.

4 Преобразования адресов

Блок сегментации транслирует логическое адресное пространство в 32-битное пространство линейных адресов. Линейный адрес образуется сложением базового адреса сегмента с эффективным адресом.

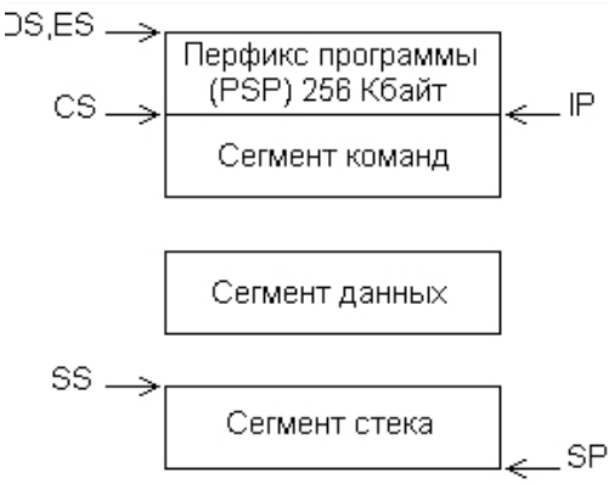

Рис. 2: Образ программы, состоящей из нескольких сегментов, в памяти компьютера

Базовый адрес сегмента в реальном режиме образуется умножением содержимого применяемого сегментного регистра на число 16_{10} (как и в процессоре i8086). В защищённом режиме базовый адрес загружается из дискриптора, хранящегося в таблице по селектору, находящемуся в используемом сегментном регистре.

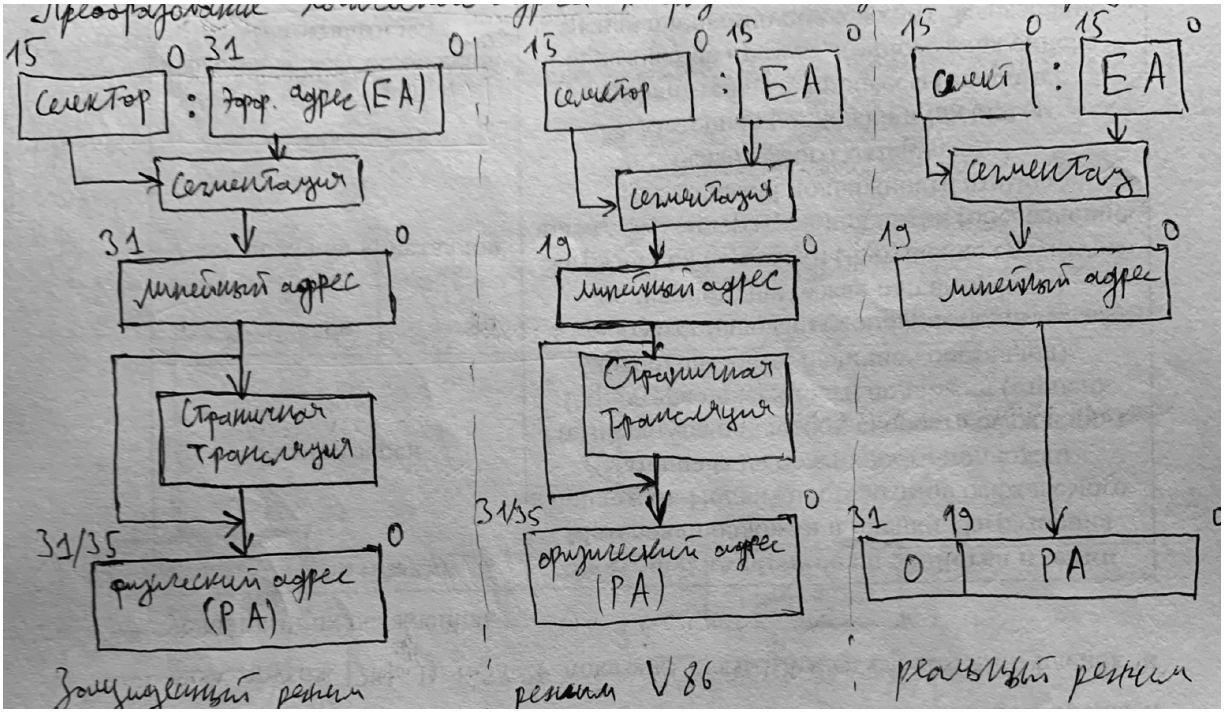

Рис. 3: Преобразования адресов в процессорах IA-32

Физический адрес памяти образуется после преобразования линейного адреса блоком страничной трансляции адресов. Он выводится на внешнюю шину адреса процессора. В простейшем случае при отключён-

ном блоке страничной трансляции физический адрес совпадает с линейным. Включённый блок страничной трансляции адресов осуществляет трансляцию линейного адреса в физический страницами различных размеров (по 4 КБайта, 2 или 4 МБайта и т. д.). Блок страничной трансляции может включаться только в защищённом режиме.

Блок страничной трансляции адресов позволяет использовать разрядность физического адреса, отличающуюся от разрядности линейного адреса. В процессорах различных моделей соотношения разрядностей менялись:

1. Процессор i80386SX при 32-битном линейном адресе имел физический адрес на 24 бита (до 16 МБайт физически адресуемой оперативной памяти).

2. В большинстве 32-битных процессоров до 6 поколения использовался 32-битный физический адрес, что позволяло подключать до 4 ГБайт физически адресуемой памяти.

3. В процессорах 6 – 8 поколений используется расширение физического адреса (PAE — Physical Address Extension): 32-битнный линейный адрес транслируется в 36-битный физический адрес, что позволяет подключать до 64 ГБайт физической адресуемой памяти.

4. Процессоры x86-64 имеют линейный адрес 48 бит, а физический — до 52 бит.

В 64-битном режиме сегментация упразднена, приложение оперирует только линейными виртуальными адресами. В 64-процессорах механизм сегментации оставлен только для режима совместимости (compatibility mode).

В режиме совместимости адреса должны соответствовать канонической форме, т. е. их старшие биты, выходящие за пределы поддерживаемой разрядности должны быть нулевыми. В противном случае сработает исключение защиты.

Из сегментных регистров процессор использует только регистры `CS`, `FS` и `GS`. В дискрипторе сегмента, на который указывает `CS`, задействуются лишь следующие атрибуты: признак 64-битного режима, размер операнда по-умолчанию и уровень привилегий. Регистры `FS` и `GS` требуются для нового режима адресации — в дискрипторе сегмента, на который они ссылаются базовый адрес, может применяться как смещение при вычислении адреса (эффективного, виртуального и линейного — теперь это одно и то же).

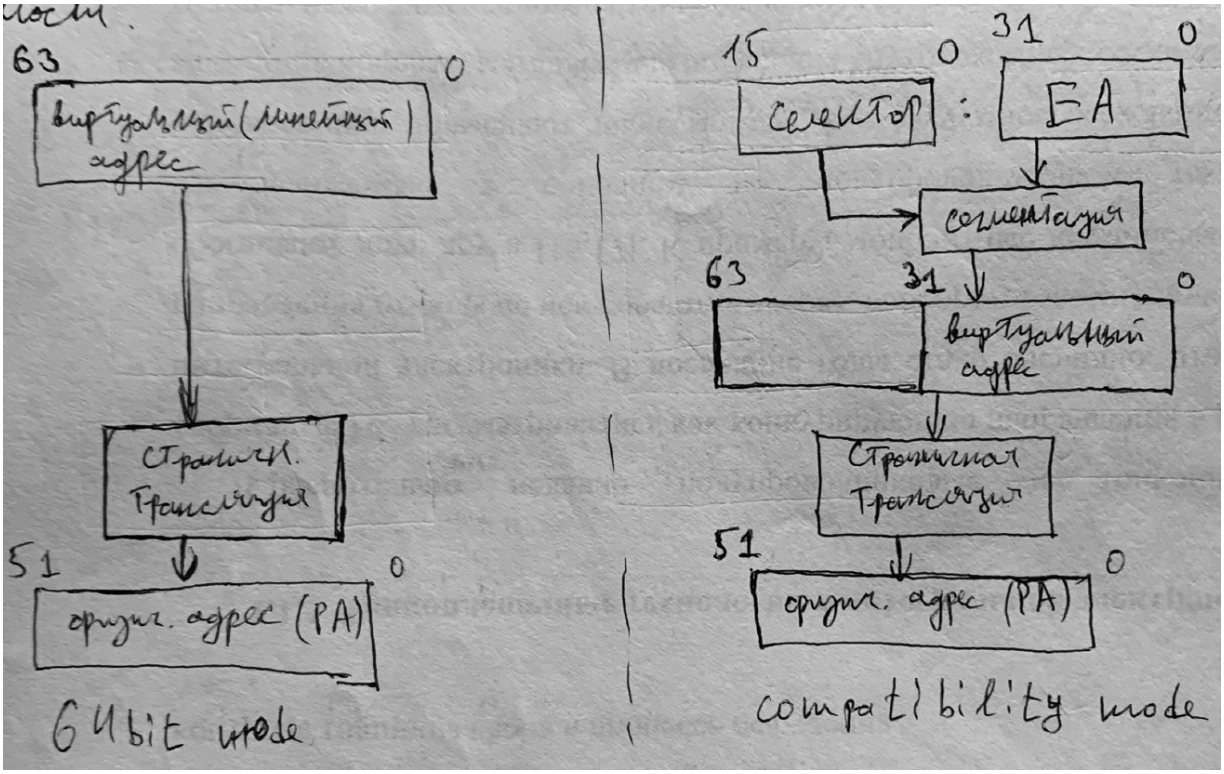

Рис. 4: Преобразования адресов в процессорах x86-64

5 Образ программы в памяти

Образ программы в памяти начинается с префикса сегмента программы (PSP — Program Segment Prefics), образуемого и заполняемого операционной системой. PSP всегда имеет размер 256 байт и содержит таблицы и поля данных, используемых системой в процессе выполнения программы. Вслед за PSP располагаются сегменты программы в том порядке, как они объявлены в программе. Сегментные регистры автоматически инициализируются следующим образом:

- `ES` и `DS` указывают на начало PSP, что даёт возможность, сохранив их содержимое, обращаться затем в программе к PSP,
- `CS` — на начало сегмента кода,
- `SS` — на начало сегмента стека.

В указатель команд `IP` загружается относительный адрес точки входа в программу, а в указатель стека `SP` — величина, равная объявленному размеру стека, в результате чего указатель стека указывает на конец стека (точнее, на первое слово за его пределами). Таким образом, после загрузки программы в память адресуемыми оказываются все сегменты, кроме сегмента данных. Инициализация регистра `DS` в первых строках программы позволяет сделать адресуемым и этот сегмент.

Важнейшая особенность архитектуры процессоров Intel — адрес любой ячейки памяти состоит из двух слов, одно из которых определяет расположение в памяти соответствующего сегмента, а другое — смещение в пределах этого сегмента.

Сегмент всегда начинается с адреса, кратного 16_{10}, т. е. на границе 16-байтового блока памяти (параграфа). Сегментный адрес можно рассматривать, как номер параграфа, с которого начинается данный сегмент. Размер сегмента определяется объёмом содержащихся в нем данных, но никогда не может превышать величину 64 КБайт, что определяется максимально возможной величиной смещения. Сегментный адрес сегмента кода хранится в регистре `CS`, а смещение адресуемого байта — в указателе команд `IP`.

Как уже отмечалось, после загрузки программы в `IP` заносится смещение первой команды программы. Процессор, считав её из памяти, увеличивает содержимое `IP` точно на длину этой команды (команды процессоров Intel могут иметь различную длину от 1 до 15 байт), в результате чего `IP` указывает на вторую команду программы. Выполнив первую команду, процессор считывает из памяти вторую, опять увеличивая значение `IP`. В результате в `IP` всегда находится смещение очередной команды, т. е. команды, следующей за выполняемой. Описанный алгоритм нарушается только при выполнении команд переходов, вызовов подпрограмм и обслуживания прерываний.

Сегментный адрес сегмента данных обычно хранится в регистре `DS`, а смещение может находится в одном из регистров общего назначения, например, в `BX` или `SI`.

6 Виды адресаций операндов инструкций в памяти

Определение 2. ***Адресацией*** *называется обращение к операнду, указание на который содержится в команде.*

Операнды, в зависимости от места своего хранения, могут указываться разными способами, которым соответствуют разные типы адресации. При описании различных адресаций операндов используют понятия адресного кода и исполнительного адреса.

Определение 3. ***Адресный код*** *— это информация об адресе операнда, содержащаяся в команде.*

Определение 4. ***Исполнительный адрес*** *— это номер физической ячейки памяти, к которой производится обращение.*

Первая группа адресаций устанавливает исполнительный адрес по значению адресного кода. Сюда входят:

- **Непосредственная адресация.** Операнд указывается в команде константой. Эта адресация используется только для указания исходных данных.

- **Регистровая адресация.** В команде указывается имя регистра процессора, в котором хранится операнд.

- **Прямая адресация.** Исполнительный адрес совпадает с адесным кодом. Прямая адресация — это простейший вид адресации операнда в памяти, так как эффективный адрес содержится в самой команде и для его формирования не используется никаких дополнительных источников или регистров. Эффективный адрес берется непосредственно из поля смещения машинной команды, которое может иметь размер 8, 16, 32 бита. Это значение однозначно определяет байт, слово или двойное слово в сегменте данных. Прямая адресация может быть двух типов.

  – **Относительная прямая адресация** используется в командах условных переходов для указания относительного адреса перехода. Относительность такого перехода заключается в том, что в поле смещения машинной команды содержится 8-, 16- или 32-разрядное значение, которое в результате работы команды будет складываться с содержимым регистра указателя команд `IP/EIP`. В результате такого сложения получается адрес, по которому и осуществляется переход.

  – **Абсолютная прямая адресация.** В этом случае эффективный адрес является частью машинной команды, но формируется этот адрес только из значения поля смещения в команде. Для формирования физического адреса операнда в памяти процессор складывает это поле со сдвинутым на четыре бита значением сегментного регистра[1]. Однако такая адресация применяется редко — обычно ячейкам в программе присваиваются символические имена. В процессе трансляции ассемблер вычисляет и подставляет значения смещений этих имен в поле смещения формируемой им машинной команды. В итоге получается, что машинная команда прямо адресует свой операнд, имея, фактически, в одном из своих полей значение эффективного адреса.

[1]Здесь имеется в виду схема образования адресов, характерная для реального режима.

- **Косвенная регистровая адресация.** Используется в целях сокращения длины команды. В этом случае адресный код указывает имя регистра процессора, в котором находится исполнительный адрес. Такой регистр называют регистром адреса. Эффективный адрес операнда может находиться в любом из регистров общего назначения, кроме `SP/ESP` и `BP/EBP` (которые зарезервированы для работы со стеком). Синтаксически в команде этот режим адресации выражается заключением имени регистра в квадратные скобки. К примеру, команда

  ```
  mov ax, [ecx]
  ```

  помещает в регистр `AX` содержимое слова по адресу сегмента данных со смещением, хранящимся в регистре `ECX`. Так как содержимое регистра легко изменить в ходе работы программы, данный способ адресации позволяет динамически назначить адрес операнда для некоторой машинной команды. Это очень полезно, например, для организации циклических вычислений и для работы с различными структурами данныхтипа таблиц или массивов.

- **Автоинкрементная и автодекрементная адресация.** В команде указывается имя регистра процессора, содержимое которого автоматически увеличивается (уменьшается) на `1`, причём изменение адреса может производиться как до (префиксный вариант), так и после (постфиксный вариант) выполнения основной команды. Следовательно, преинкремент/предекремент означает вычисление нового исполнительного адреса перед выполнением команды, а для постинкремента/постдекремента исполнительный адрес в данной команде не изменяется.

Вторая группа адресаций устанавливает исполнительный адрес по адресному коду и содержимому дополнительных регистров процессора. Сюда входят **индексная** и **базовая** адресации. Оба типа адресаций позволяют при меньшей длине адресного кода команды обеспечить доступ к любой ячейке памяти.

- **Индексация** означает автоматическое изменение исполнительного адреса без изменения содержимого регистра адреса, называемого индексом, причем исполнительный адрес вычисляется как алгебраическая сумма содержимого индекса и смещения. Таким образом, содержимое индекса задает начало некоторой области ячеек памяти, а смещение — конкретную ячейку памяти в этой области. В

команде адресный код указывает имя индекса и сравнительно короткое смещение или имя регистра процессора, в котором оно содержится. Индексация используется при работе с массивами данных.

- **Базирование** является развитием индексации. Здесь исполнительный адрес также определяется алгебраической суммой содержимого регистра адреса, называемого базой, и смещения, но с изменением содержимого базы. При этом используются постинкремент/постдекремент и преинкремент/предекремент на величину смещения.

Замечание 1. *Все виды адресации, кроме непосредственной, регистровой и прямой относятся к косвенным. Слово косвенный в названии этих видов адресации означает, что в самой команде может находиться лишь часть эффективного адреса, а остальные его компоненты находятся в регистрах.*

Комбинируя все перечисленные выше виды косвенной адресации, можно получить следующие варианты.

Косвенная базовая адресация со смещением предназначена для доступа к данным с известным смещением относительно некоторого базового адреса. Этот вид адресации удобно использовать для доступа к элементам структур данных, когда смещение элементов известно заранее на стадии разработки программы, а базовый (начальный) адрес структуры должен вычисляться динамически на стадии выполнения программы. Модификация содержимого базового регистра позволяет обращаться к одноименным элементам различных экземпляров однотипных структур данных.

К примеру, команда

```
mov ax, [edx + 3]
```

пересылает в регистр `AX` слово из области памяти по адресу, определяемому содержимым `EDX+3`.

Косвенная индексная адресация со смещением очень похожа на косвенную базовую адресацию со смещением. Здесь также для формирования эффективного адреса используется один из регистров общего назначения. Но индексная адресация обладает одной интересной особенностью, которая очень удобна для работы с массивами. Она связана с возможностью масштабирования содержимого индексного регистра.

Косвенная базовая индексная адресация. Эффективный адрес формируется как сумма содержимого двух регистров общего назначения: базового и индексного. В качестве этих регистров могут применяться любые регистры общего назначения, при этом часто содержимое индексного регистра масштабируется.

Косвенная базовая индексная адресация со смещением. Эффективный адрес формируется как сумма трёх составляющих: содержимого базового регистра, содержимого индексного регистра и значения поля смещения в команде.

7 Модели управления памятью IA-32

В семействе процессоров IA-32 выбор метода обращения к памяти определяется режимом работы процессора. Возможны три основных режима работы.

- Реальный. В этом режиме адрес формируется аналогично i8086, т. е. при формировании адреса используются 16-ти разрядные смещения и 16-ти разрядные сегментные адреса, которые хранятся в сегментных регистрах. При их сложении по приведенной выше схеме получаются 20-ти разрядные физические адреса, поэтому в этом режиме доступен только первый мегабайт оперативной памяти. Реальный режим работы процессора используется в операционной системе MS DOS, которая устарела. В настоящее время этот режим не используется в прикладных программах и применяется только на этапе первоначальной загрузки компьютера.

- Защищенный. В этом режиме используется 32-х разрядная адресация, предусматривающая несколько вариантов защиты, откуда и появилось название этого режима.

- Виртуальный. В этом режиме процессор моделирует псевдоодновременную работу нескольких виртуальных процессоров i8086. В настоящее время режим также устарел и практически не используется.

Кроме перечисленных выше трёх основынх режимов существует ещё режим системного управления (SMM — System Management Mode) — это режим работы процессора, впервые появившийся в процессоре Pentium. Он обеспечивает операционную систему механизмом для выполнения машинно-зависимых функций, таких как перевод компьютера в режим пониженного энергопотребления или выполнения действий по защите системы. Для перехода в данный режим процессор должен получить специальный сигнал `SMI` от усовершенствованного программируемого контроллера прерываний (APIC — Advanced Programmable Interrupt Controller), при этом сохраняется состояние вычислительной среды процессора. Функционирование процессора в этом режиме подобно его рабо-

те в режиме реальной адресации. Возврат из этого режима производится специальной командой процессора.

Реальный режим. Ранняя ОС MS-DOS работала в реальном режиме, в то время как более поздние (OS/2, Windows, Linux и др.) обычно требуют защищённого режима. После включения питания (в процессе загрузки) процессор стартует в реальном режиме и затем начинает загружать программы в оперативную память из ПЗУ и с диска. Для того чтобы переключить процессор в защищённый режим, соответствующая программа должна находиться в последовательности команд начальной загрузки.

Защищённый режим. Кроме очевидной дополнительной адресуемости памяти выше предела 1 Мбайт для DOS, здесь имеется также ряд других преимуществ:

- защищённая память, которая препятствует взаимным помехам программ;
- виртуальная память, или выделение дополнительного адресного пространства на жёстком диске так, чтобы программы использовали больше оперативной памяти, чем физически установлено на машине;
- переключение задач (многозадачный режим), когда несколько задач выполняются квазиодновременно (путем быстрого и своевременного переключения активности процессора от одной задачи к другой).

Размер памяти в защищённом режиме обычно ограничивается 4 Гбайт, что, однако, не окончательный предел размера памяти в процессорах IA-32. Используя такие приёмы, как страничная память и управление сегментами памяти, процессор IA-32 под управлением ОС может обратиться более чем к 32-битовому адресному пространству, даже без переключения к 64 битам (один из этих приемов известен как физическое расширение адреса — Physical Address Extension).

Кроме того, можно использовать виртуальный режим 8086 — это подрежим защищённого режима, являющийся неким гибридом, который позволяет старым программам DOS выполняться под управлением супервизора защищённого режима, а также поддерживает одновременное выполнение программ в Protected mode и DOS. В то время как защищённый режим появился уже в 16-битном процессоре 80286, виртуальный возник только в версии защищённого режима IA-32.

7.1 Реальный режим процессоров i80x86

Рассмотрим далее основные принципы функционирования процессоров i80x86 в реальном режиме.

Оперативную память при работе в этом режиме можно разбить на логические блоки по 64 КБайт, называемые сегментами, причём каждый сегмент может начинаться с адреса, кратного 16 Байт. Таким образом, первый сегмент имеет начальный адрес `0`, второй находится по адресу `16` (или `10h`) и т. д. Несколько близко расположенных сегментов могут перекрываться. Это удобно при организации совместного доступа к командам и данным разными программами.

Доступ к каждой ячейке в памяти происходит путём указания значения регистра сегмента и смещения — адреса внутри этого блока. Например, команда, подлежащая исполнению процессором в каждый данный момент времени, определяется из значений двух регистров — регистра `CS`, значение которого, будучи умноженным на 16, даёт адрес начала сегмента команд, и регистра указателя команд `IP` (Instruction Pointer), указывающего положение соответствующей команды относительно начала сегмента команд.

7.1.1 Оперативная память

Объём оперативной памяти i80x86 (здесь имеется в виду 8086) — 1 МБайт (2^{20}). Байты нумеруются, начиная с `0`, и номер байта называется его адресом. Для ссылок на байты памяти используются 20-разрядные адреса — от `00000h` до `FFFFFh`.

Байт содержит восемь разрядов (битов), каждый из которых может принимать значение `1` или `0`. Разряды нумеруются справа налево от `0` до `7`. Байт — наименьшая адресуемая ячейка памяти — используется для хранения небольших целых чисел и символов.

В i80x86 используются и более крупные ячейки — слова, двойные, учетверённые и т. п. слова. Слово — два соседних байта, размер — 16 бит (они нумеруются справа налево от `0` до `15`). Используется для хранения целых чисел и адресов. Адресом слова считается адрес его первого байта (с меньшим адресом); этот адрес может быть чётным и нечётным. Двойное слово — четыре соседних байта (два соседних слова), размер — 32 бита. Адресом двойного слова считается адрес его первого байта. Используется для хранения «длинных» целых[2] чисел и так называемых адресных пар (`SEGMENT:OFFSET`).

[2]Условное понятие, так как обычно в языке C двойные слова используются для хранения переменных типа `int`, в то время как действительно длинные (`long int`) переменные представляются учетверёнными словами

7.1.2 Представление символов и строк

На символ отводится один байт памяти, в который записывается код символа — целое от `0` до `255`. В i80x86 используется система кодировки ASCII.

Строка (последовательность символов) размещается в соседних байтах памяти (в неперевернутом виде): код первого символа строки записывается в первом байте, код второго символа — во втором байте и т. п. Адресом строки считается адрес её первого байта.

В i80x86 строкой считается также и последовательность слов (обычно это последовательность целых чисел). Элементы таких строк располагаются в последовательных ячейках памяти, но каждый элемент представлен в «перевернутом» виде.

7.1.3 Представление адресов

Адрес — это порядковый номер ячейки памяти, т. е. неотрицательное целое число, поэтому в общем случае адреса представляются так же, как и числа без знака. Однако в i80x86 есть ряд особенностей в представлении адресов.

Дело в том, что в i80x86 термином «адрес» могут обозначаться разные понятия:

- 16-битовое смещение (offset) — адрес ячейки, отсчитанный от начала сегмента (области) памяти, которому принадлежит эта ячейка. В этом случае под адрес отводится слово памяти, причем адрес записывается в «перевернутом» виде (как и числа-слова вообще);

- 20-битовый абсолютный адрес некоторой ячейки памяти. В силу ряда причин в i80x86 такой адрес задается не как 20-битовое число, а как пара `SEGMENT:OFFSET`, где `SEGMENT` — это 16 бит начального адреса сегмента памяти, которому принадлежит ячейка, а `OFFSET` (смещение) — 16-битовый адрес этой ячейки, отсчитанный от начала данного сегмента памяти (величина `10h*SEGMENT+OFFSET` даёт абсолютный адрес ячейки).

  Такая пара записывается в виде двойного слова: в первом слове размещается смещение, а во втором — сегмент, причём каждое из этих слов в свою очередь представлено в «перевернутом» виде. Например, пара `1234h:5678h` будет записана так, как показано на рис. 5.

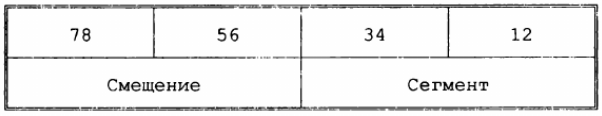

Рис. 5:

7.1.4 Директивы определения данных

Для того чтобы в программе на NASM зарезервировать ячейки памяти под константы и переменные, необходимо воспользоваться директивами определения данных — с названиями `DB` (описывает данные размером в байт), `DW` (слово), `DD` (двойное слово), `DQ` (учетверённое слово), `DT` (пятикратное слово — 80 бит), `DDQ` или `DO` (восьмикратное слово).

Директивы, или команды ассемблеру — это предложения программы, которыми её автор сообщает какую-то информацию ассемблеру или предписывает что-либо сделать дополнительно, помимо перевода символьных команд на машинный язык.

В простейшем случае в директиве `DB`, `DW` или `DD` описывается одна константа, которой даётся имя для последующих ссылок на неё. По этой директиве ассемблер формирует машинное представление константы (в частности, если надо, «переворачивает» её) и записывает в очередную ячейку памяти. Адрес этой ячейки становится значением имени: все вхождения имени в программу ассемблер будет заменять на этот адрес.

Имена, указанные в директивах `DB`, `DW` и `DD`, называются именами переменных (в отличие от меток — имен команд).

В NASM числа записываются в нормальном (неперевернутом) виде в системах счисления с основанием 10 (по умолчанию), 16, 8 или 2.

Пример 1. *Директивы определения данных в NASM.*

```
A DB 162    ; описать константу-байт 162 и дать ей имя A
B DB 0A2h   ; такая же константа, но с именем B
C DW -1     ; константа-слово -1 с именем C
D DW 0FFFFh ; такая же константа-слово, но с именем D
E DD -1     ; константа -1 с именем E как двойное слово
```

Константы-символы описываются в директиве `DB` двояко: указывается либо код символа (целое от `0` до `255`), либо сам символ в кавычках (одинарных или двойных). Например, следующие директивы эквивалентны (`2A` — код звездочки в ASCII).

Пример 2. *Директивы определения символьных данных в NASM.*

```
CH1 DB 02Ah
```

```
CH2 DB '*'
CH3 DB "*"
```

В одной директиве можно описать сразу несколько констант и/или переменных одного и того же размера. С этой целью их следует перечислить через запятую. Они размещаются в соседних ячейках памяти.

Пример 3. *Перечисления данных — аналог массивов.*

```
array1 dw -7, 132, 0, 1097
array2 dd 1, 2, 3, 4, 5
```

Имя, указанное в директиве, считается именующим первую из констант. Для ссылок на остальные в NASM используются выражения вида `<имя> + <целое>`. Так, для доступа к двойному слову с числом 4 в предыдущем примере надо использовать выражение `array2 + 3` и т. д.

Если в директиве `DB` перечислены только символы, тогда эту директиву можно записать короче, конкатенируя их все в одну строку.

Пример 4. *Строки как перечисления байтов.*

```
string db 'Hello, world!', 0Ah, 0Dh, '$'
```

Замечание 2. *Любой размер больше, чем `DD` не допускает строк в качестве операндов.*

Неинициализированные данные [3] можно объявить в программе с помощью директив NASM `RESB`, `RESW`, `RESD`, `RESQ`, `REST`, `RESDQ` или `RESO`. Каждая из этих директив должна сопровождаться одним операндом, являющимся числом резервируемых байт, слов, двойных слов и т. д.

Пример 5. *Примеры объявления неинициализированных данных.*

```
buffer      resb 64 ; резервирование 64 байт
shortvar    resw 1  ; резервирование слова
doublearray resq 10 ; массив из 10 чисел с плавающей точкой
```

Замечание 3. *NASM не поддерживает синтаксис резервирования неинициализированного пространства, реализованный в MASM/TASM, где можно делать `DW ?` или подобные вещи: это полностью упразднено.*

[3] Данные такого типа размещаются в BSS-секции объектного модуля.

Префикс `TIMES` заставляет инструкцию ассемблироваться несколько раз. Данная псевдо-инструкция отчасти представляет NASM-эквивалент синтаксиса `DUP`, поддерживающегося MASM-совместимыми ассемблерами. Однако `TIMES` более разносторонняя инструкция, т. к. её аргумент — не просто числовая константа, а числовое выражение.

Пример 6. *Использование префикса `TIMES`.*

```
zerobuf: times 64 db 0

buffer: db 'Hello, world'
        times 64-$+buffer db ' '
```

Во втором случае будет зарезервировано строго определённое пространство, расширяющее полную длину `buffer` до 64 байт.

Константы-адреса, как правило, задаются именами. Так, по директиве

```
ADR1 DW CH1
```

будет выделено слово в памяти, которому даётся имя `ADR1` и в которое будет помещен адрес (смещение), соответствующий имени `CH1`.

Если использовать директиву `SEG` как показано ниже, то ассемблер NASM добавит к смещению имени его сегментную часть адреса.

```
ADR2 DW CH2, SEG CH2
```

7.1.5 Представление команд. Модификация адресов

Структура команд. Машинные команды i80x86 занимают от 1 до 6 Байтов. Код операции (OpCODE) занимает один или два первых байта команды. В i80x86 достаточно много различных операций, поэтому для них не хватает 256 различных кодов операций, которые можно представить в одном байте. В связи с этим некоторые операции объединяются в группу и им даётся один и тот же OpCODE, во втором же байте команды он уточняется. Кроме того, во втором байте указываются типы и способ адресации операндов. Остальные байты команды указывают на операнды.

Исполнительные адреса. Команды могут иметь от 0 до трёх операндов, у большинства команд — один или два операнда. Размер операндов — байт или слово (редко — двойное слово). Операнд может быть указан в самой команде (это так называемый непосредственный операнд) либо может находиться в одном из регистров i80x86 и тогда в команде указывается этот регистр, либо может находиться в ячейке памяти и тогда в команде тем или иным способом указывается адрес этой ячейки. Некоторые команды требуют, чтобы операнд находился в фиксированном месте (например, в регистре `AX`), тогда операнд явно не указывается в команде. Результат выполнения команды помещается в регистр или ячейку памяти, из которого (которой), как правило, берётся первый операнд.

Адрес операнда разрешено модифицировать по одному или двум регистрам.

В первом случае в качестве регистра-модификатора разрешено использовать регистры `BX`, `BP`, `SI` или `DI` (и никакие иные).

Во втором случае один из модификаторов обязан быть регистром `BX` или `BP`, а другой — регистром `SI` или `DI`. Одновременная модификация по `BX` и `BP` или `SI` и `DI` недопустима. Регистры `BX` и `BP` обычно используются для хранения базы (или базового, начального адреса) некоторого участка памяти (например, массива) и потому называются базовыми регистрами, а регистры `SI` и `DI` часто содержат индексы элементов массива и потому называются индексными регистрами. Однако такое «распределение ролей» необязательно, и, например, в `SI` может находиться база массива, а в `BX` — индекс элемента массива.

При выполнении команды процессор прежде всего вычисляет так называемый исполнительный (эффективный) адрес — сумму адреса, заданного в команде, и текущих значений указанных регистров-модификаторов

Полученный таким образом 16-разрядный адрес определяет смещение — адрес, отсчитанный от начала некоторого сегмента (области) памяти. Перед обращением к памяти процессор добавляет к смещению начальный адрес этого сегмента (он хранится в некотором сегментном регистре) и в результате получается окончательный 20-разрядный адрес, по которому происходит реальное обращение к памяти.

7.1.6 Сегментирование

Процессор Intel 8080 использовал 16-битную шину адреса, и поэтому поддерживал оперативную память объемом не более 64 КБайт (2^{16}). В процессоре Intel модели 8086 применялась уже 20-битная шина адреса, что позволило увеличить адресуемую память до 1 Мбайт (2^{20}), а в i80286 — 24-битный адрес (и до 16 МБайт памяти, соответственно). Од-

нако в этих моделях процессоров семейства x86 ради обратной совместимости были сохранены 16-битные адреса — именно такие адреса хранятся в регистрах, указываются в командах и образуются в результате модификации по базовым и индексным регистрам.

Проблема согласования различных разрядностей внутренних регистров процессора и его внешней шины адреса решается с помощью сегментирования (неявного базирования) адресов. В i8086 абсолютный (20-битный) адрес `A` любой ячейки памяти можно представить как сумму 20-битового начального адреса (базы) `B` сегмента, которому принадлежит ячейка, и 16-битового смещения `D` — адреса этой ячейки, отсчитанного от начала сегмента:

$$A = B + D. \tag{1}$$

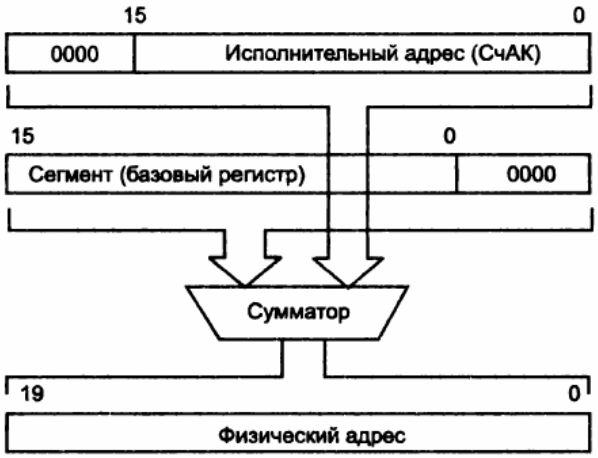

Рис. 6: Модификация адресов при сегментной адресации

Неоднозначность выбора сегмента не играет существенной роли, главное — чтобы сумма `B` и `D` давала нужный адрес). Адрес `B` заносится в некоторый регистр `S`, а в команде, где должен быть указан адрес `A`, вместо него записывается пара из регистра `S` и смещения `D` (в NASM такая пара, называемая адресной парой или указателем, записывается как `S:D`).

Процессор же устроен так, что при выполнении команды он прежде всего по паре `S:D` вычисляет абсолютный адрес `A` как сумму содержимого регистра `S` и смещения `D` и только затем обращается к памяти по адресу `A`. Заменяя в командах абсолютные адреса на адресные пары, удается адресовать всю память 16-битными адресами (смещениями).

В качестве регистра `S` разрешается использовать не любой регистр, а только один из четырех регистров, называемых сегментными: `CS`, `DS`, `SS` и `ES`. В связи с этим одновременно можно работать с четырьмя сегментами памяти: начало одного из них загружается в регистр `CS` и все ссылки на ячейки этого сегмента указываются в виде пар `CS:D`, начало другого

заносится в DS и все ссылки на его ячейки задаются в виде пар DS:D и т. д. Если одновременно надо работать с большим числом сегментов, тогда следует своевременно сохранять содержимое сегментных регистров и записывать в них начальные адреса пятого, шестого и других сегментов.

Отметим, что используемые сегменты могут быть расположены в памяти произвольным образом: они могут не пересекаться, а могут пересекаться и даже совпадать. Какие сегменты памяти использовать, в каких сегментных регистрах хранить их начальные адреса — всё это решение програмиста.

Как и все регистры i8086, сегментные регистры имеют размер слова. Поэтому возникает проблема — как удается разместить в них 20-битовые начальные адреса сегментов памяти? Решение следующее — поскольку все эти адреса кратны 16, то в них младшие 4 бита (последняя шестнадцатеричная цифра) всегда нулевые, а потому эти биты можно не хранить явно, а лишь подразумевать. Именно так и происходит — в сегментном регистре всегда хранятся только старшие 16 битов (старшие четыре шестнадцатеричные цифры) начального адреса сегмента (эта величина называется номером сегмента или просто сегментом). При вычислении же абсолютного адреса A по паре S:D процессор сначала приписывает справа к содержимому регистра S четыре нулевых бита (другими словами, умножает на 16) и лишь затем прибавляет смещение D, причем суммирование ведется по модулю 2^{20}:

$$A_{\text{abs}} = S \cdot 10_{16} + D \pmod{2^{20}}. \tag{2}$$

Если, например, в регистре CS хранится величина 1234h, тогда адресная пара 1234h:507h определяет абсолютный адрес, равный 12340h + 507h = 12847h.

Сегментные регистры по умолчанию. Согласно описанной схеме сегментирования адресов, замену абсолютных адресов на адресные пары надо производить во всех командах, имеющих операнд-адрес. Однако был разработан способ, позволяющий избежать указания этих пар в большинстве команд, суть которого состоит в том, что по умолчанию устанавливается, какой именно сегментный регистр соответствует данному сегменту памяти, а в командах задается только смещение. Явно не указанный сегментный регистр автоматически подставляется в команду, и только при необходимости нарушить эти умолчания следует полностью указывать адресную пару. Список умолчаний приводится ниже.

С учетом такого распределения ролей сегментных регистров машинные программы обычно строятся так: все команды программы размещаются в одном сегменте памяти, начало которого заносится в регистр CS,

Регистр	Умолчания	Комментарий
CS	Указывает на начало области памяти, в которой размещены команды программы (эта область называется сегментом команд или сегментом кодов), и потому при ссылках на ячейки этой области регистр CS можно не указывать явно, он подразумевается по умолчанию	Абсолютный адрес очередной команды, подлежащей выполнению, всегда задается парой CS:IP; в счетчике команд IP всегда находится смещение этой команды относительно адреса из регистра CS
DS	Указывает на сегмент данных (область памяти с константами, переменными и другими величинами программы)	Во всех ссылках на этот сегмент регистр DS можно явно не указывать, так как он подразумевается по умолчанию
SS	Указывает на стек — область памяти, доступ к которой осуществляется по принципу «последним записан — первым считан»	Все ссылки на стек, в которых явно не указан сегментный регистр, по умолчанию сегментируются по регистру SS
ES	Считается свободным, он не привязан ни к какому сегменту памяти и его можно использовать по своему усмотрению	Чаще всего он применяется для доступа к данным, которые не поместились или сознательно не были размещены в сегменте данных

Рис. 7: Сегментные регистры по умолчанию

а все данные размещаются в другом сегменте, начало которого заносится в регистр `DS`. Если нужен стек, то под него отводится третий сегмент памяти, начало которого записывается в регистр `SS`. После этого практически во всех командах можно указывать не полные адресные пары, а лишь смещения, так как сегментные регистры в этих парах будут восстанавливаться автоматически.

Здесь возникает следующий вопрос: как по смещению определить, на какой именно сегмент памяти оно указывает? В общих чертах ответ такой — ссылки на сегмент команд могут содержаться только в командах перехода, а ссылки практически во всех других командах (кроме строковых и стековых) — это ссылки на сегмент данных. Например, в команде пересылки

```
mov ax, x
```

имя `x` воспринимается как ссылка на данное, а потому автоматически восстанавливается до адресной пары `DS:x`. В команде же безусловного перехода по адресу, находящемуся в регистре `BX`,

```
jmp bx
```

абсолютный адрес перехода определяется парой `cs:[bx]`.

Если в ссылке на какую-то ячейку памяти не указан явно сегментный регистр, то этот регистр берется по умолчанию. Явно же сегментные регистры надо указывать, только если по каким-то причинам регистр по умолчанию не подходит. Если, например, в команде пересылки нам

надо сослаться на стек (скажем, надо записать в регистр `AH` байт стека, помеченный именем `x`), тогда уже будет недостаточно предположения о том, что по умолчанию операнд команды `MOV` будет сегментироваться по регистру `DS`, и потому следует явно указать иной регистр — в данном случае регистр `SS`, так как именно он указывает на стек:

```
MOV AH, SS:X.
```

Однако такие случаи встречаются редко и потому в командах, как правило, указывается только смещение.

Отметим, что в NASM сегментный регистр записывается в самой команде непосредственно перед смещением (именем переменной, меткой и т. п.), однако на уровне машинного языка ситуация несколько иная. Предусмотрено четыре специальные однобайтовые команды, именуемые префиксами замены сегмента (обозначаемые как `CS:`, `DS:`, `SS:` и `ES:`). Они ставятся перед командой, операнд-адрес которой должен быть просегментирован по регистру, отличному от регистра, подразумеваемого по умолчанию. Например, приведенная ранее символическая команда пересылки — это на самом деле две машин ные команды:

```
SS:
MOV AH, X
```

Сегментирование, базирование и индексирование адресов. Поскольку сегментирование адресов — это разновидность модификации адресов, то в i8086 адрес, указываемый в команде, в общем случае модифицируется по трём регистрам — сегментному, базовому и индексному. В целом модификация адреса производится в два этапа. Сначала учитываются только базовый и индексный регистры (если они, конечно, указаны в команде), причём вычисление здесь происходит в области 16-битовых адресов; полученный в результате 16-битовый адрес называется исполнительным (или эффективным) адресом. Если в команде не предусмотрено обращение к памяти (например, она загружает адрес в регистр), то на этом модификация адреса заканчивается и используется именно исполнительный адрес (он загружается в регистр).

Если же нужен доступ к памяти, тогда на втором этапе исполнительный адрес рассматривается как смещение, и к нему прибавляется (умноженное на 10_{16}) содержимое сегментного регистра, указанного явно или взятого по умолчанию, и в результате вычисляется абсолютный (физический) 20-битовый адрес, по которому фактически происходит обращение к памяти (см. рис. 6).

Отметим, что сегментный регистр учитывается только непосредственно перед обращением к памяти, а до этого работа ведется только с 16-битовыми адресами. Если учесть к тому же, что сегментные регистры, как правило, не указываются в командах, то можно считать, что i8086 работает с 16-битовыми адресами.

Как уже сказано, если в ссылке на ячейку памяти не указан сегментный регистр, то он определяется по умолчанию. Эго делается по следующим правилам:

Программные сегменты. Директива ASSUME. Рассмотрим, как сегментирование проявляется в программах на MASM. Для того чтобы указать, что некоторая группа операторов программы образует единый сегмент памяти, они оформляются как про граммный сегмент: перед ними ставится директива segment, по сле них — директива ends , причем в начале обеих этих директив должно быть указано одно и то же имя, играющее роль имени сегмента. Программа же в целом представляет собой последова тельность таких программных сегментов, в конце которой ука зывается директива конца программы end , например: dti segment; программный сегмент с именем DT1 A DB O B DW ? DTI ENDS DT2 segment; программный сегмент DT2 C DB ' h e l lo ' DT2 ENDS Г code segment; программный сегмент code ASSUME CS:CODE, DS:DT1, ES:DT2 PROG: MOV AX,DT2 MOV DS,AX MOV BH,C CODE ENDS end prog; конец текста программы. Предложения программного сегмента ассемблер размещает в одном сегменте памяти (в совокупности они не должны зани мать более 64 Кбайт), начиная с ближайшего свободного адреса, кратного 16. Номер (первые 16 битов начального адреса) этого сегмента становится значением имени сегмента. В MASM это имя относится к константным выражениям, а не адресным, по этому в команде MOV AX,DT2, второй операнд является непосредственным, поэтому в регистр ax будет записано начало (номер) сегмента DT2, а не содержи мое начальной ячейки этого сегмента. Имена же переменных (A, в, с) и метки (prog) относятся к адресным выражениям, и им ставится в соответствие адрес их ячейки относительно «своего» сегмента: • имени A соответствует адрес 0; • имени в — адрес 1; • имени с — адрес 0, а метке prog — адрес 0. Все ссылки на предложения одного программного сегмента ассемблер сегментирует по умолчанию по одному и тому же сег ментному регистру, по какому именно — устанавливается спе циальной директивой assume . В приведенном примере эта ди ректива определяет, что все ссылки на сегмент code, если явно не указан сегментный регистр, должны сегментироваться по ре гистру CS, все ссылки на DT1 — по регистру DS, а все ссылки на DT2 — по регистру ES. Встретив в тексте программы ссылку на какое-либо

имя (на пример, на имя c в команде mov ax, c), ассемблер определяет, в каком именно программном сегменте оно описано (здесь в DT2), затем по информации из директивы assume «узнает», ка кой сегментный регистр поставлен в соответствие этому сегменту (в данном случае — это ES), и далее образует адресную пару из данного регистра и смещения имени (здесь — ES : 0), которую и за писывает в формируемую машинную команду. При этом ассемб лер учитывает используемое в i80x86 соглашение о сегментных ре гистрах по умолчанию — если в адресной паре, построенной им самим или явно заданной в программе, сегментный регистр совпа дает с регистром по умолчанию, то в машинную команду заносит ся лишь смещение. Если, скажем, встретится команда MOV cx,в, тогда по имени в ассемблер построит пару D S:l, но если опе ранд — адрес команды mov — по умолчанию сегментируется по регистру DS, то записывать этот регистр в машинную команду излишне, и ассемблер записывает в нее только смещение 1. Таким образом, директива a s s u m e избавляет программистов от необходимости выписывать полные адресные пары не только тогда, когда используются сегментные регистры по умолчанию (как в случае с именем в), но тогда, когда в машинной команде следовало бы явно указать сегментный регистр (как в случае с именем c). В MASM сегментный регистр в ссылке на имя требу ется указывать лишь тогда, когда имя должно по каким-либо причинам сегментироваться по регистру, отличному от того, что поставлен в соответствие всему сегменту, в котором это имя описано. Однако все это справедливо только при соблюдении следую щих условий: • директива a s s u m e должна быть указана перед первой ко мандой программы. В противном случае ассемблер, «про сматривающий» текст программы сверху вниз, не будет знать, как сегментировать имена из команд, расположен ных до этой директивы, и потому зафиксирует ошибку; • в директиве a s s u m e следует каждому сегменту ставить в со ответствие сегментный регистр: если ассемблеру встретится ссылка на имя из сегмента, которому не соответствует ника кой сегментный регистр, то он зафиксирует ошибку. Прав да, в обоих случаях можно избежать ошибки, но для этого в ссылках необходимо явно указывать сегментный регистр.

Начальная загрузка (инициализация) сегментных регистров. Сделать такую загрузку — обязанность самой программы, с загрузки сегментных регистров и должно начинаться выполнение программы. Поскольку в i8086 нет команды пересылки непосредственного операнда в сегментный регистр (а имя, т. е. начало, сегмента — это непосредственный операнд), то такую загрузку приходится делать через какой-то другой, несегментный, регистр (например, `AX`):

MOV AX,DT1 ;AX := начало сегмента DT1 MOV DS,AX ;DS := A X
.

Аналогично загружается регистр e s .

Загружать регистр `CS` в начале программы не надо: он, как и счётчик команд `IP`, загружается операционной системой перед тем, как начинается выполнение программы (иначе нельзя было бы начать её выполнение). Что же касается регистра `SS`, используемого для работы со стеком, то он может быть загружен так же, как и регистры `DS` и `ES`, однако в NASM предусмотрена возможность загрузки этого регистра ещё до выполнения программы (и об этом написано в разделе 8).

Ссылки вперед. «Встречая» в команде ссылку назад — имя, которое описано в тексте программы до этой команды, ассемб лер уже имеет необходимую информацию об имени и потому может правильно оттранслировать эту команду. Но если встре тится ссылка вперед или имя, которое не было описано до ко манды и которое, вероятно, будет описано позже, то ассемблер в большинстве случаев не сможет правильно оттранслировать та кую команду. Например, не «зная», в каком программном сег менте будет описано это имя, ассемблер не может установить, по какому именно сегментному регистру надо сегментировать имя, и потому не может определить, надо или же нет размещать перед соответствующей машинной командой префикс замены сегмен та и, если надо, то какой именно.

В подобной ситуации ассемблер действует следующим образом:

• если в команде встретилась ссылка вперед, то он делает не которое предположение относительно этого имени и уже на основе этого предположения формирует машинную ко манду; • если затем (когда встретится описание имени) окажется, что данное предположение было неверным, тогда ассемб лер пытается исправить сформированную ранее машинную команду. Однако это не всегда удается: если правильная машинная команда должна занимать больше места, чем машинная команда, построенная на основе предположения (например, перед командой надо на самом деле вставить префикс замены сегмента), тогда ассемблер фиксирует ошибку (как правило, это ошибка номер 6: Phase error between passes).

Какие же «предположения» делает ассемблер, встречая ссыл ку вперед?

Во всех командах, кроме команд перехода, ассемблер «пред полагает», что имя будет описано в сегменте данных и потому сегментируется по регистру d s . Это следует учитывать при составлении программы: если в команде встречается ссылка вперед на имя, которое описано в сегменте, на начало которого указы вает сегментный регистр, отличный от DS, то перед таким име нем автор программы должен написать соответ-

ствующий пре- фике. Пример: CODE SEGMENT ASSUME CSrCODE X DW ? BEG: MOV AX, X ; здесь вместо CS: X можно записать просто X MOV CS: Y, AX ; здесь обязательно надо записать CS: Y Y DW ? CODE ENDS.

p 378

p 383

p 384

7.2 Защищённый режим

Требование сохранить возможность выполнения программ, использующих 16-ти разрядную адресацию, привело к тому, что схема 32-х разрядной адресации является многокомпонентной.

В этом режиме по-прежнему используется сегментная организация памяти, но размер сегмента уже не ограничивается 64 Кб, а теоретически может достигать 4 Гб. 32-х разрядный адрес базы сегмента хранится не в виде сегментного ад- реса в сегментном регистре, как при 16-ти разрядной адресации, а полностью в специальных внутренних регистрах процессора – дескрипторах. Номер дескрип- тора заносится в 14 бит сегментного регистра, который в этом режиме называ- ется селектором. Один бит селектора из этих 14-ти отвечает за выбор таблицы локальных или глобальных дескрипторов.

p 109

Таблица локальных дескрипторов содержит дескрипторы сегментов приложе- ния, а таблица глобальных – дескрипторы сегментов программ операционной системы. Оставшиеся два бита селектора содержат код уровня привилегий сег- мента, который проверяется при обращениях из других программ. Таким обра- зом, реализуется защита сегментов.

14 бит селектора и 32 бита эффективного или исполнительного адреса, форми- руемого на основе машинной команды, объединяются в 46-ти разрядный вирту- альный адрес.

Сумма 32-х разрядного базового адреса сегмента и 32-х разрядного эффектив- ного адреса образует 32-х разрядный линейный адрес. Физический же адрес определяется по таблице страниц на основе линейного.

Соответственно различают несколько адресных пространств: виртуальное – 64 Тб; линейное – 4 Гб; физическое – 4 Гб.

При создании приложений Windows в основном используется модель памяти Flat «плоская». Эта модель подразумевает, что каждому приложению отводится линейное адресное пространство объемом 2 Гб, а остальные 2 Гб предостав- ляются операционной системе. Базовый адрес в дескрипторах всех сегментов приложения устанавливается равным 0. В

езультате все сегменты приложения «перекрываются». Программа, данные и стек размещаются в разных местах памяти за счет азличных смещений. Разделение памяти между приложениями осуществляется операционной системой, которая размещает страницы прило- жений с одинаковыми линейными адресами в разных местах оперативной па- мяти. Следовательно и защита сегментов при этой модели не работает.

Чтобы предотвратить взаимное влияние выполняющихся программ друг на дру- га им выделяются изолированные участки памяти (т.е. код и данные программ находятся во взаимно несмежных сегментах). В защищенном режиме работают такие ОС, как MS Windows и Linux.

В типичной программе, написанной для защищенного режима есть 3 сегмента: кода, данных и стека, информация о которых хранится в трех перечисленных ниже сегментных регистрах. В регистре CS хранится указатель на дескриптор сегмента кода програм- мы. В регистре DS хранится указатель на дескриптор сегмента данных про- граммы. В регистре SS хранится указатель на дескриптор сегмента стека програм- мы.

3.4.4. Сегментные регистры В программной модели микропроцессора имеется 6 сегментных регистров: CS, SS, DS, ES, GS, FS.

Их существование обусловлено спецификой организации и использования опе- ративной памяти микропроцессорами Intel. Она заключается в том, что микро- процессор аппаратно поддерживает структурную организацию программы в виде 3 частей, называемых сегментами. Соответственно, такая организация памяти называется сегментной.

Для того чтобы указать на сегменты, к которым программа имеет доступ в кон- кретный момент времени, и предназначены сегментные регистры. Фактически в этих регистрах содержатся адреса памяти, с которых начинаются соответст- вующие сегменты. Логика обработки машинной команды построена так, что при выборке команды, доступе к данным программы или к стеку неявно используют- ся адреса во вполне определенных сегментных регистрах. Микропроцессор поддерживает следующие типы сегментов:

Эти регистры используются в качестве базовых при обращении к заранее рас- пределенным областям оперативной памяти, которые называются сегментами. Существует 3 типа сегментов и, соответственно, сегментных регистров: кода (CS), в них хранятся только команды процессора, т.е. машинный код программы; данных (DS, ES, FS и GS). В них хранятся области памяти, выделяемые под переменные программы и под данные; стека SS), в них хранится системная область памяти, называемая стеком, в которой распределяются локальные (временные) переменные програм- мы и параметры, передаваемые функциям при их вызове.

Сегмент кода. Содержит команды программы. Для доступа к этому

сегменту служит регистр CS (code segment register) — сегментный регистр кода. Он со- держит адрес начала сегмента с машинными командами, к которому имеет дос- туп микропроцессор (то есть эти команды загружаются в конвейер микропро- цессора).

Суть сегментной адресации заключается в следующем. Обращение к памяти осуществляется исключительно с помощью сегментов - логических образова- ний, накладываемых на те или иные участки физической памяти. Исполнитель- ный адрес любой ячейки памяти вычисляется процессором путем сложения начального адреса сегмента, в котором располагается эта ячейка, со смещени- ем к ней (в байтах) от начала сегмента. Это смещение иногда называют относи- тельным адресом. Образование физического адреса из сегментного адреса и смещения:

р 122

Начальный адрес сегмента без четырех младших битов, т.е. деленный на 16, помещается в один из сегментных регистров и называется сегментным адре- сом. Сам же начальный адрес хранится в специальном внутреннем регистре процессора, называемом теневым регистром. Для каждого сегментного регист- ра имеется свой теневой регистр; начальный адрес сегмента загружается в него процессором в тот момент, когда программа заносит в соответствующий сег- ментный регистр новое значение сегментного адреса.

Сегмент данных. Содержит обрабатываемые программой данные. Для доступа к этому сегменту служит регистр ds (data segment register) — сегментный ре- гистр данных, который хранит адрес начала сегмента данных текущей програм- мы.

Сегмент стека. Этот сегмент представляет собой область памяти, называемую стеком. Работу со стеком микропроцессор организует по принципу LIFO (Last In First Out – последним пришел, первым ушел). Для доступа к этому сегменту служит регистр SS (stack segment register) — сегментный регистр стека, содер- жащий адрес начала сегмента стека. Для работы со стеком в системе команд микропроцессора есть специальные команды, а в программной модели микро- процессора для этого существуют специальные регистры:

С помощью регистра ESP/SP (Stack Pointer register) происходит обраще- ние к данным, хранящимся в стеке. Этот регистр обычно никогда не ис- пользуется для выполнения обычных арифметических операций и команд пересылки данных. Его часто иазывают расширенным регистром указате- ля стека (extended stack pointer).

Регистр EBP/BP (Base Pointer register) обычно используется для произ- вольной адресации в стеке параметров и локальных переменных. Регистр ESP - указатель стека, автоматически модифицируется коман-

дами PUSH, POP, RET, CALL. Явно используется реже.

Дополнительный сегмент данных. Неявно алгоритмы выполнения большин- ства машинных команд предполагают, что обрабатываемые ими данные распо- ложены в сегменте данных, адрес которого находится в сегментном регистре ds.

Если программе недостаточно одного сегмента данных, то она имеет возмож- ность использовать еще 3 дополнительных сегмента данных. Но в отличие от основного сегмента данных, адрес которого содержится в сегментном регистре ds, при использовании дополнительных сегментов данных их адреса требуется указывать явно с помощью специальных префиксов переопределения сегмен- тов в команде. Адреса дополнительных сегментов данных должны содержаться в регистрах es, gs, fs (extension data segment registers).

8 Стек. Подпрограммы

Стеком называют область программы для временного хранения произвольных данных, доступ к которым осуществляется по принципу «последним записан — первым считан». Во многих случаях программе требуется временно запомнить информацию, а затем считывать её в обратном порядке. Эта проблема как раз и решается посредством реализации стека LIFO (Last Input — First Output), называемого также стеком включения/извлечения (stack — кипа, стопа).

Разумеется, данные можно сохранять и в сегменте данных, однако в этом случае для каждого сохраняемого на время данного надо заводить отдельную именованную ячейку памяти, что увеличивает размер программы и количество используемых имен. Удобство стека заключается в том, что его область используется многократно, причем сохранение в стеке данных и выборка их оттуда выполняется с помощью эффективных команд `PUSH` и `POP` без указания каких-либо имен.

Наиболее важное использование стека связано с процедурами.

- Стек традиционно используется, например, для сохранения содержимого всех регистров (инструкция `PUSHA`), используемых программой, перед вызовом подпрограммы, которая, в свою очередь, может использовать регистры процессора «в своих личных целях». Исходное содержимое регистров извлекается из стека после возврата из подпрограммы инструкцией `POPA`.

- Другой распространенный приём — передача подпрограмме требуемых ею параметров через стек. Подпрограмма, зная, в каком

порядке помещены в стек параметры, может забрать их оттуда и использовать при своем выполнении.

В системе команд i80x86 имеются специальные инструкции для работы со стеком, но для того чтобы можно было ими воспользоваться, необходимо соблюдение ряда условий.

Стек обычно рассчитан на косвенную адресацию через регистр SP — указатель стека. При включении элементов в стек производится автоматический декремент указателя стека, а при извлечении — инкремент, т. е. стек всегда «растет» в сторону меньших адресов памяти. Адрес последнего включенного в стек элемента называется вершиной стека (TOS — Top Of Stack).

Под стек можно отвести область в любом месте памяти. Размер её может быть любым, но не должен превосходить 64 КБайт, а начальный адрес должен быть кратным 16. Другими словами, эта область должна быть сегментом памяти, который называется сегментом стека. Начало этого сегмента (первые 16 битов начального адреса) должно храниться в сегментном регистре SS.

Хранимые в стеке элементы могут иметь любой размер, однако следует учитывать, что в i80x86 имеются команды записи в стек и чтения из него только слов. Поэтому для записи байта в стек его надо предварительно расширить до слова, а запись или чтение двойных слов осуществляются парой команд.

В i80x86 принято заполнять стек снизу вверх, от больших адресов к меньшим: первый элемент записывается в конец области, отведенной под стек, второй элемент — в предыдущую ячейку области и т. д. Считывается всегда элемент, записанный в стек последним. В связи с этим нижняя граница стека всегда фиксирована, а верхняя — меняется. Слово памяти, в котором находится элемент стека, записанный последним, называется вершиной стека. Адрес вершины, отсчитанный от начала сегмента стека, обязан находиться в указателе стека — регистре SP. Таким образом, абсолютный адрес вершины стека определяется парой SS:SP.

Значение «0» в регистре SP свидетельствует о том, что стек полностью заполнен (его вершина «дошла» до начала области стека). Поэтому для контроля за переполнением стека надо перед новой записью в стек проверять условие SP = 0 (процессор i80x86 этого не делает).

Для пустого стека значение SP должно равняться размеру стека, т. е. пара SS:SP должна указывать на байт, следующий за последним байтом области стека. Другими словами, в исходном состоянии SP указывает на ячейку, лежащую под дном стека и не входящую в него. Контроль за чтением из пустого стека, если надо, обязана делать сама программа.

Начальная установка регистров SS и SP может быть произведена в

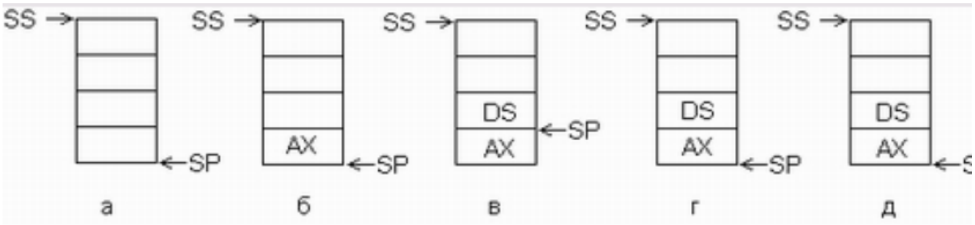

Рис. 8: Состояния стека: а — исходное состояние, б — после загрузки одного элемента (в данном примере — содержимого регистра `AX`), в — после загрузки второго элемента (содержимого регистра `DS`), г — после выгрузки одного элемента, д — после выгрузки двух элементов и возврата в исходное состояние

самой программе, однако в NASM предусмотрена возможность автоматической загрузки этих регистров. Для этого при описании сегмента стека директивой `SEGMENT` необходимо указать его класс, как показано в следующем примере.

Пример 7. *Рекомендуемый вариант описания сегмента стека программы в ассемблере NASM.*

```
segment stack class=stack
  resb 512
```

В этом случае ассемблер (точнее — загрузчик) перед тем, как передать управление на первую команду машинной программы, автоматически загрузит в регистры `SS` и `SP` нужные значения. Так, если под сегмент стека из примера выше была выделена область памяти, начиная с абсолютного адреса `1234h`, то к началу выполнения программы в регистре `SS` окажется величина `1234h`, а в регистре `SP` — значение `200h` (т. е. 512_{10}). Отметим, что все эти значения соответствуют пустому стеку.

Пример 8. *Вариант описания сегмента стека с самостоятельной его настройкой программистом.*

```
segment code
  mov ax, stack
  mov ss, ax
  mov sp, stacktop

segment stack
  resb 512
stacktop:
```

Замечание 4. *Приведённый выше пример также работоспособен, но на этапе компоновки программы, линкер может выводить автоматические предупреждения (warnings) о том, что сегмент стека программы не обнаружен.*

Основные стековые команды. При соблюдении указанных выше требований в программе можно использовать команды, предназначенные для работы со стеком. Основными из них являются следующие.

- Запись слова в стек:

  `PUSH op`

  Здесь `op` обозначает любой 16-битный регистр (в том числе и сегментный) или адрес слова памяти. По этой команде значение регистра `SP` уменьшается на `2` (вычитание происходит по модулю 2^{16}), затем указанное операндом слово записывается в стек по адресу `SS:SP`.

- Чтение слова из стека:

  `POP op`

  Слово, считанное из вершины стека, присваивается операнду `op` (регистру, в том числе сегментному, но не `CS`, или слову памяти), затем значение `SP` увеличивается на `2`.

- Переход с возвратом:

  `CALL op`

  Эта команда записывает адрес следующей за ней команды в стек и затем делает переход по адресу, определяемому операндом `op`. Она используется для переходов на подпрограммы с запоминанием в стеке адреса возврата. Имеются следующие разновидности этой команды (они аналогичны вариантам команды безусловного перехода `JMP`).

  – Внутрисегментный относительный длинный переход (`op` — непосредственный операнд размером в слово, а в NASM — это метка из текущего сегмента кода или имя близкой процедуры). В этом случае в стек заносится только текущее значение счетчика команд `IP`, т. е. смещение следующей команды.

  - Внутрисегментный абсолютный косвенный переход (`op` — адрес слова памяти, в которой находится адрес (смещение) той команды, на которую и будет сделан переход). И здесь в стек записывается только смещение адреса возврата.
  - Межсегментный абсолютный прямой переход (`op` — непосредственный операнд вида `SEG:OFS`, а в NASM — это
  - Межсегментный абсолютный косвенный переход (`op` — адрес двойного слова, в котором находится пара `SEG:OFS`, задающая абсолютный адрес перехода). И здесь в стеке сохраняется содержимое регистров `CS` и `IP`.

- Переход (возврат) по адресу из стека:

  `RET op`

  Из стека считывается адрес и по нему производится переход. Если указан операнд (а это должно быть неотрицательное число), то после чтения адреса стек дополнительно очищается на это число байтов (к `SP` добавляется это число). Команда `RET` используется для возврата из подпрограммы по адресу, записанному в стек командой `CALL` при вызове подпрограммы, и одновременной очистки стека от параметров, которые основная программа занесла в стек перед обращением к подпрограмме.

Совсем не обязательно при восстановлении данных из стека помещать их туда, где они были перед сохранением. Например, можно поместить в стек содержимое `DS`, а извлечь его оттуда в другой сегментный регистр — `ES`

```
push DS
pop ES
```

Теперь `ES = DS`, а стек пуст. Это распространенный приём для перенесения содержимого одного регистра в другой, особенно, если второй регистр — сегментный.

Замечание 5. *После выгрузки сохраненных в стеке данных, они не стираются физически, а остаются в области стека на своих местах. Однако, при «стандартной» работе со стеком они оказываются недоступными. Действительно, поскольку указатель стека* ***SP*** *указывает под дно стека, стек считается пустым. Очередная команда* ***push*** *поместит новое данное на место сохраненного ранее содержимого, затерев его. Однако пока стек физически не затерт, сохраненными и уже*

выбранными из него данными можно пользоваться, если помнить, в каком порядке они расположены в стеке. Этот приём часто используется при работе с подпрограммами.

Какого размера должен быть стек? Это зависит от того, насколько интенсивно он используется в программе. Если, например, планируется хранить в стеке массив объемом 10 000 Байт, то стек должен быть не меньше этого размера. При этом надо иметь в виду, что в ряде случаев стек автоматически используется ОС, в частности, при выполнении команды прерывания `int 21h`. По этой команде сначала процессор помещает в стек адрес возврата, а затем операционная система отправляет туда же содержимое регистров и другую информацию, относящуюся к прерванной программе. Поэтому, даже если программа совсем не использует стек, он все же должен присутствовать в программе и иметь размер не менее нескольких десятков (а лучше — сотен) машинных слов.

Если для стека определён слишком маленький сегмент, то возможно его переполнение (stack overflow). Переполнение в ОС защищённого режима вызывает срабатывание защиты. В ОС реального режима переполнение стека приводит к “загадочным” вылетам и зависаниям программы. Переполнение может происходить при интенсивных прерываниях, когда до завершения обработки одного прерывания возникает и обрабатывается другое, более приоритетное прерывание.