Введение

Есть высокоуровневые языки — это те, где вы говорите if — else, print, echo, function и так далее. «Высокий уровень» означает, что вы говорите с компьютером более-менее человеческим языком. Другой человек может не понять, что именно у вас написано в коде, но он хотя бы сможет прочитать слова.

Но сам компьютер не понимает человеческий язык. Компьютер — это регистры памяти, простые логические операции, единицы и нули. Поэтому прежде чем ваша программа будет исполнена процессором, ей нужен переводчик — программа, которая превратит высокоуровневый язык программирования в низкоуровневый машинный код.

Ассемблер — это собирательное название языков низкого уровня: код всё ещё пишет человек, но он уже гораздо ближе к принципам работы компьютера, чем к принципам мышления человека.

Вариантов Ассемблера довольно много. Но так как все они работают по одинаковому принципу и используют (в основном) одинаковый синтаксис, мы будем все подобные языки называть общим словом «Ассемблер».

Язык ассемблера — это символическое представление машинного языка. Все процессы в персональном компьютере (ПК) на самом низком, аппаратном уровне приводятся в действие только командами (инструкциями) машинного языка. По-настоящему решить проблемы, связанные с аппаратурой (или даже, более того, зависящие от аппаратуры как, к примеру, повышение быстродействия программы), невозможно без знания ассемблера.

Ассемблер представляет собой удобную форму команд непосредственно для компонент ПК и требует знание свойств и возможностей интегральной микросхемы, содержащей эти компоненты, а именно микропроцессора ПК. Таким образом, язык ассемблера непосредственно связан с внутренней организацией ПК. И не случайно практически все компиляторы языков высокого уровня поддерживают выход на ассемблерный уровень программирования.

Элементом подготовки программиста-профессионала обязательно является изучение ассемблера. Это связано с тем, что программирование на ассемблере требует знание архитектуры ПК, что позволяет создавать более эффективные программы на других языках и объединять их с программами на ассемблере.

Что такое ассемблер?

Само слово **ассемблер** (assembler) переводится с английского как «сборщик». На самом деле так называется программа-транслятор, принимающая на входе текст, содержащий условные обозначения машинных команд, удобные для человека, и переводящая эти обозначения в последовательность соответствующих кодов машинных команд, понятных процессору. В отличие от машинных команд, их условные обозначения, называемые также **мнемониками**, запомнить сравнительно легко, так как они представляют собой сокращения от английских слов. В дальнейшем мы будем для простоты именовать мнемоники ассемблерными командами. Язык условных обозначений и называется **языком ассемблера**.

На заре компьютерной эры первые ЭВМ занимали целые комнаты и весили не одну тонну, имея объем памяти с воробьиный мозг, а то и того меньше. Единственным способом программирования в те времена было вбивать программу в память компьютера непосредственно в цифровом виде, переключая тумблеры, проводки и кнопочки. Число таких переключений могло достигать нескольких сотен и росло по мере усложнения программ. Встал вопрос об экономии времени и денег. Поэтому следующим шагом в развитии стало появление в конце сороковых годов прошлого века первого транслятора-ассемблера, позволяющего удобно и просто писать машинные команды на человеческом языке и в результате автоматизировать весь процесс программирования, упростить, ускорить разработку программ и их отладку. Затем появились языки высокого уровня и **компиляторы** (более интеллектуальные генераторы кода с более понятного человеку языка) и **интерпретаторы** (исполнители написанной человеком программы на лету). Они совершенствовались, совершенствовались — и, наконец, дошло до того, что можно просто программировать мышкой.

Таким образом, ассемблер — это **машинно ориентированный язык** программирования, позволяющий работать с компьютером напрямую, один на один. Отсюда и его полная формулировка — язык программирования низкого уровня второго поколения (после машинного кода). Команды ассемблера один в один соответствуют командам процессора, но поскольку существуют различные модели процессоров со своим собственным набором команд, то, соответственно, существуют и разновидности, или диалекты, языка ассемблера. Поэтому использование термина «язык ассемблера» может вызвать ошибочное мнение о существовании единого языка низкого уровня или хотя бы стандарта на такие языки. Его не существует. Поэтому при именовании языка, на котором написана конкретная программа, необходимо уточнять, для какой архитектуры она предназначена и на каком диалекте языка написана. Поскольку ассемблер привязан к устройству процессора, а тип процессора жестко определяет набор доступных команд машинного языка, то программы на ассемблере не переносимы на иную компьютерную архитектуру.

Поскольку ассемблер всего лишь программа, написанная человеком, ничто не мешает другому программисту написать свой собственный ассемблер, что часто и происходит. На самом деле не так уж важно, язык какого именно ассемблера изучать. Главное — понять сам принцип работы на уровне команд процессора, и тогда не составит труда освоить не только другой ассемблер, но и любой другой процессор со своим набором команд.

Как мыслит процессор

Чтобы понять, как работает Ассемблер и почему он работает именно так, нам нужно немного разобраться с внутренним устройством процессора.

Кроме того, что процессор умеет выполнять математические операции, ему нужно где-то хранить промежуточные данные и служебную информацию. Для этого в самом процессоре есть специальные ячейки памяти — их называют регистрами.

Регистры бывают разного вида и назначения: одни служат, чтобы хранить информацию; другие сообщают о состоянии процессора; третьи используются как навигаторы, чтобы процессор знал, куда идти дальше, и так далее.

Так вот: всё, с чем работает Ассемблер, — это команды процессора, переменные и регистры.

Здесь нет привычных типов данных — у нас есть только байты памяти, в которых можно хранить что угодно. Даже если вы поместите в ячейку какой-то символ, а потом захотите работать с ним как с числом — у вас получится. А вместо привычных циклов можно просто прыгнуть в нужное место кода.

Команды Ассемблера

Каждая команда Ассемблера — это команда для процессора. Не операционной системе, не файловой системе, а именно процессору — то есть в самый низкий уровень, до которого может дотянуться программист.

Любая команда на этом языке выглядит так:

```
[&lt;метка&gt;:] &lt;команда&gt; [&lt;операнды&gt;] [;&lt;комментарий&gt;]
```

Метка — это имя для фрагмента кода. Например, вы хотите отдельно пометить место, где начинается работа с жёстким диском, чтобы было легче читать код. Ещё метка нужна, чтобы в другом участке программы можно было написать её имя и сразу перепрыгнуть к нужному куску кода.

Команда — служебное слово для процессора, которое он должен выполнить. Специальные компиляторы переводят такие команды в машинный код. Это сделано для того, чтобы не запоминать сами машинные команды, а использовать вместо них какие-то буквенные обозначения, которые проще запомнить. В этом, собственно, и выражается человечность Ассемблера: команды в нём хотя бы отдалённо напоминают человеческие слова.

Операнды отвечают за то, что именно будут делать команды: какие ячейки брать для вычислений, куда помещать результат и что сделать с ним дополнительно. Операндом могут быть названия регистров, ячейки памяти или служебные части команд.

Комментарий — это просто пояснение к коду. Его можно писать на любом языке, и на выполнение программы он не влияет. Примеры команд:

```
<strong>mov eax, ebx</strong> ; Пересылаем значение регистра EBX в регистр
EAX
```

```
<strong>mov x, 0</strong> ; Записываем в переменную x значение 0
<strong>add eax, x</strong> ; Складываем значение регистра EAX и переменной
x, результат отправится в регистр EAX
```

Здесь нет меток, первыми идут команды (mov или add), а за ними — операнды и комментарии.

Пример: возвести число в куб

Если нам понадобится вычислить x^3, где x занимает ровно один байт, то на Ассемблере это будет выглядеть так.

Первый вариант

```
<strong>mov   al, x</strong> ; Пересылаем x в регистр AL
<strong>imul  al</strong> ; Умножаем регистр AL на себя, AX = x * x
<strong>movsx bx, x</strong> ; Пересылаем x в регистр BX со знаковым
расширением
<strong>imul  bx</strong> ; Умножаем AX на BX. Результат разместится в DX:AX
```

Второй вариант

```
<strong>mov   al, x</strong> ; Пересылаем x в регистр AL
<strong>imul  al</strong> ; Умножаем регистр AL на себя, AX = x * x
<strong>cwde</strong> ; Расширяем AX до EAX
<strong>movsx ebx, x</strong> ; Пересылаем x в регистр EBX со знаковым
расширением
<strong>imul  ebx</strong> ; Умножаем EAX на EBX. Поскольку x – 1-байтовая
переменная, результат благополучно помещается в EAX
```

На любом высокоуровневом языке возвести число в куб можно одной строкой. Например:

x = Math.pow (x, 3);

x: = exp (ln (x) * 3);

на худой конец x = x*x*x.

Хитрость в том, что когда каждая из этих строк будет сведена к машинному коду, этого кода может быть и 5 команд, и 10, и 50, и даже 100. Чего стоит вызов объекта Math и его метода pow: только на эту служебную операцию (ещё до самого возведения в куб) может уйти несколько сотен и даже тысяч машинных команд.

А на Ассемблере это гарантированно пять команд. Ну, или как реализуете.

Почему это круто

Ассемблер позволяет работать с процессором и памятью напрямую — и делать это очень быстро. Дело в том, что в Ассемблере почти не тратится зря процессорное время. Если процессор работает на частоте 3 гигагерца — а это примерно 3 миллиарда процессорных команд в секунду, — то очень хороший код на Ассемблере будет выполнять примерно 2,5 миллиарда команд в секунду. Для сравнения, JavaScript или Python выполнят в тысячу раз меньше команд за то же время.

Ещё программы на Ассемблере занимают очень мало места в памяти. Именно поэтому на этом языке пишут драйверы, которые встраивают прямо в устройства, или управляющие программы, которые занимают несколько килобайт. Например, программа, которая находится в брелоке сигнализации и управляет безопасностью всей машины, занимает всего пару десятков килобайт. А всё потому, что она написана для конкретного процессора и использует его возможности на сто процентов.

Справедливости ради отметим, что современные компиляторы C++ дают машинный код, близкий по быстродействию к Ассемблеру, но всё равно немного уступают ему.

Почему это сложно

Для того, чтобы писать программы на Ассемблере, нужно очень любить кремний:

- понимать архитектуру процессора;
- знать устройство железа, которое работает с этим процессором;
- знать все команды, которые относятся именно к этому типу процессоров;
- уметь работать с данными в побайтовом режиме (забудьте о строках и массивах, ведь ваш максимум — это одна буква);
- понимать, как в ограниченных условиях реализовать нужную функциональность.

Теперь добавьте к этому отсутствие большинства привычных библиотек для работы с чем угодно, сложность чтения текста программы, медленную скорость разработки — и вы получите полное представление о программировании на Ассемблере.

Для чего всё это

Ассемблер незаменим в таких вещах:

- драйверы;
- программирование микроконтроллеров и встраиваемых процессоров;
- куски операционных систем, где важно обеспечить скорость работы;
- антивирусы (и вирусы).

На самом деле на Ассемблере можно даже запилить свой сайт с форумом, если у программиста хватает квалификации. Но чаще всего Ассемблер используют там, где даже скорости и возможностей C++ недостаточно.

Ассемблер — программирование или искусство?

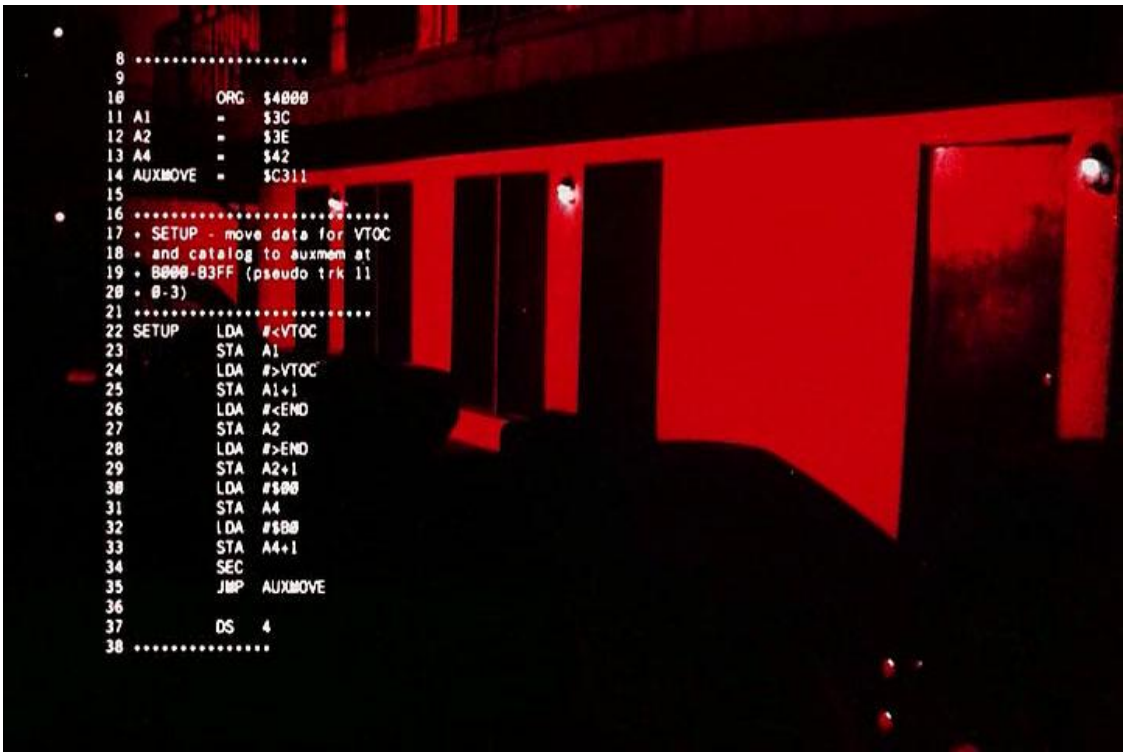
Скажем так, все зависит от того, в чьих руках он находится. Ассемблер — это первичный элемент мира процессора, из сочетаний этих элементов складывается его душа, его самосознание. Подобно тому, как вся музыка, написанная в истории человечества, состоит из сочетаний семи нот, так и сочетание ассемблерных команд наполняет компьютерный мир цифровой жизнью. Кто знает лишь три аккорда — это «попса», кому же известна вся палитра — это классика.

Почему же наука так жаждет проникнуть в квантовые глубины и захватить в свои руки неуловимый первичный кирпичик материи? Чтобы получить над ней власть, изменять ее по своей воле, стать на уровень Творца Вселенной. В чьи руки попадет такая власть — это еще вопрос. В отличие от науки, в мире программирования тайн нет, нам известны кирпичики, его составляющие, а следовательно, и та власть над процессором, которую нам дает знание ассемблера.

Чтобы программирование на языке ассемблера поднялось на уровень искусства, нужно постичь его красоту, скрывающуюся за потоком единиц и нулей. Как и в любой отрасли человеческой деятельности, в программировании можно быть посредственностью, а можно стать Мастером. И то и другое отличает степень культуры, образования, труда и, главное, то, сколько души автор вкладывает в свое творение.

Ассемблер и терминатор

Не так давно Джеймс Кэмерон выпустил в свет 3D-версию второго «Терминатора», и в качестве интересного исторического факта можно отметить один любопытный момент из жизни киборга-убийцы...



Кадр из фильма «Терминатор»

Здесь мы видим «зрение» терминатора, а слева на нем отображается ассемблерный листинг. Судя по нему, знаменитый Уничтожитель работал на процессоре MOS Technology 6502 либо на MOS Technology 6510. Этот процессор впервые был разработан в 1975 году, использовался на компьютерах Apple и, помимо всего прочего, на знаменитых игровых приставках того времени Atari 2600 и Nintendo Entertainment System (у нас более известной как Dendy). Имел лишь три 8-разрядных регистра: A-аккумулятор и два индексных регистра X и Y. Такое малое их количество компенсировалось тем, что первые 256 байт оперативной памяти (так называемая нулевая страница) могли адресоваться специальным образом и фактически использовались в качестве 8-разрядных или 16-разрядных регистров. У данного процессора было 13 режимов адресации на всего 53 команды. У терминатора идет цепочка инструкций LDA-STA-LDA-STA... В семействе 6502 программы состояли чуть менее чем полностью из LDA/LDY/LDX/STA/STX/STY:

```
LDA — загрузить в аккумулятор
LDY — загрузить в регистр Y
LDX — загрузить в регистр X
STA — сохранить из аккумулятора
STX — сохранить из регистра X
STY — сохранить из регистра Y
```

Чтение и запись в порты ввода-вывода также выполнялись этими командами, и программа терминатора имеет вполне осмысленный вид, а не представляет собой бестолковую фантазию сценариста: MOS Technology 6502 / Система команд.

Отрасли практического применения

Ранее упоминалось, что в наше время ассемблер почти вытеснен языками высокого уровня. Однако и по сей день ему находится применение. Приведем некоторые примеры.

- Разработка встроенного программного обеспечения. Это небольшие программы, не требующие значительного объема памяти на таких устройствах, как, например, телефоны, автомобильные системы зажигания, системы безопасности, видео- и звуковые карты, модемы и принтеры. Ассемблер для этого идеальный инструмент.
- В компьютерных игровых консолях для оптимизации и уменьшения объема кода и для быстродействия.
- Для использования в программе новых команд, доступных на новых процессорах. Компилятор высокого уровня хоть и оптимизирует код при компиляции, но практически никогда не способен генерировать инструкции из расширенных наборов команд типа AVX, CTV, XOP. Потому что команды в процессоры добавляют быстрее, чем в компиляторах появляется логика для генерации этих команд.
- Большая доля программ для графического процессора GPU пишется на ассемблере, наряду с языками высокого уровня HLSL или GLSL.
- Для написания кода, создание которого невозможно или затруднено на языках высокого уровня, например получение дампа памяти/стека. Даже когда аналог на языке высокого уровня возможен, преимущество языка ассемблера может быть значительным. Например, реализация подсчета среднего арифметического двух чисел с учетом переполнения для x86 процессоров занимает всего две команды (сложение с выставлением флага переноса и сдвиг с займом этого флага). Аналог на языке высокого уровня `((long) x + y) >> 1` либо может не работать в

принципе, ведь `sizeof(long) == sizeof(int)`, либо при компиляции конвертируется в огромнейшее количество команд процессора.

- Написание вирусов и антивирусников. Единственный язык программирования для создания достойных инфекторов — CIH, Sality, Sinowal.
- И конечно же, нельзя не упомянуть оборотную сторону медали: взлом, крэкинг и более легальный вариант — reverse engineering. Знание ассемблера — это мощнейший инструмент в руках реверсера. Ни дизассемблирование, ни отладка программ без знаний о нем невозможны.
-

---

1. Архитектура ПК.

Архитектура ЭВМ – это абстрактное представление ЭВМ, которое отражает ее структурную, схемотехническую и логическую организацию.

Все современные ЭВМ обладают некоторыми общими и индивидуальными свойствами архитектуры. Индивидуальные свойства присущи только конкретной модели компьютера.

Понятие архитектуры ЭВМ включает в себя:

- структурную схему ЭВМ;
- средства и способы доступа к элементам структурной схемы ЭВМ;
- набор и доступность регистров;
- организацию и способы адресации;
- способ представления и формат данных ЭВМ;
- набор машинных команд ЭВМ;
- форматы машинных команд;
- обработка прерываний.

Основные элементы аппаратных средств компьютера: системный блок, клавиатура, устройства отображения, дисководы, печатающие устройства (принтер) и различные средства связи. Системный блок состоит из системной платы, блока питания и ячеек расширения для дополнительных плат. На системной плате размещены микропроцессор, постоянная память (ROM), оперативная память (RAM) и сопроцессор.

1. Регистры.

Внутри микропроцессора информация содержится в группе из 32 регистров (16 пользовательских, 16 системных), в той или иной мере доступных для использования программистом. Так как пособие посвящено программированию для микропроцессора 8088-i486, то логичнее всего начать эту тему с обсуждения внутренних регистров микропроцессора, доступных для пользователя.

Пользовательские регистры используются программистом для написания программ. К этим регистрам относятся:

- восемь 32-битных регистров (регистры общего назначения) EAX/AX/AH/AL, EBX/BX/BH/BL, ECX/CX/CH/CL, EDX/DX/DLH/DL, EBP/BP, ESI/SI, EDI/DI, ESP/SP;
- шесть 16 -,битовых регистров сегментов: CS,DS, SS, ES, FS,GS;
- регистры состояния и управления: регистр флагов EFLAGS/FLAGS, и регистр указателя команды EIP/IP.

Через наклонную черту приведены части одного 32-разрядного регистра. Приставка E (Extended) обозначает использование 32-разраядного регистра. Для работы с байтами используются регистры с приставками L (low) и H(high), например, AL,CH - обозначающие младший и старший байты 16-разрядных частей регистров.

1. Регистры общего назначения.

EAX/AX/AH/AL(Accumulator register) –*аккумулятор*. Используются при умножении и делении, в операциях ввода-вывода и в некоторых операциях над строками.

EBX/BX/BH/BL – *базовый регистр* (base register), часто используется при адресации данных в памяти.

ECX/CX/CH/CL – *счетчик* (count register), используется как счетчик числа повторений цикла.

EDX/DX/DH/DL –*регистр данных* (data register), используется для хранения промежуточных данных. В некоторых командах использование его обязательно.

Все регистры этой группы позволяют обращаться к своим «младшим» частям. Использование для самостоятельной адресации можно только младшие 16- и 8-битовые части этих регистров. Старшие 16 бит этих регистров как самостоятельные объекты недоступны.

Для поддержки команд обработки строк, позволяющих производить последовательную обработку цепочек элементов имеющих длину 32, 16 или 8 бит используются:

ESI/SI (source index register) – *индекс источника*. Содержит адрес текущего элемента источника.

EDI/DI (distination index register) – *индекс приемника* (получателя). Содержит текущий адрес в строке приемнике.

В архитектуре микропрцессора на программно-аппаратном уровне поддерживается структура данных – стек. Для работы со стеком есть специальные команды и специальные регистры. Следует отметить, что стек заполняется в сторону меньших адресов.

ESP/SP (stack poINTer register) –*регистр указателя стека*. Содержит указатель вершины стека в текущем сегменте стека.

EBP/BP (base poINTer register) –*регистр указателя базы стека*. Предназначен для организации произвольного доступа к данным внутри стека.

1.1.2. Сегментные регистры

В программной модели микропроцессора имеются шесть *сегментных регистров:* CS, SS, DS, ES, GS, FS. Их существование обусловлено спецификой организации и использования оперативной памяти микропроцессорами Intel. Микропроцессор аппаратно поддерживает структурную организацию программы состоящей из *сегментов.* Для указания сегментов доступных в данный момент предназначены сегментные регистры. Микропроцессор поддерживает следующие типы сегментов:

1. *Сегмент кода.* Содержит команды программы Для доступа к этому сегменту служит регистр CS (code segment register) – *сегментный регистр кода*. Он содержит адрес сегмента с машинными командами, к которому имеет доступ микропроцессор.

2. *Сегмент данных.* Содержит обрабатываемые программой данные. Для доступа к этому сегменту служит регистр DS (data segment register) – *сегментный регистр данных*, который хранит адрес сегмента данных текущей программы.
3. *Сегмент стека.* Этот сегмент представляет собой область памяти, называемую стеком. Микропроцессор организует стек по принципу – первый «пришел», первый «ушел». Для доступа к стеку служит регистр SS (stack segment register) – *сегментный регистр стека*, содержащий адрес сегмента стека.
4. *Дополнительный сегмент данных.* Обрабатываемые данные могут находиться еще в трех дополнительных сегментах данных. По умолчанию предполагается, что данные находятся в сегменте данных. При использовании дополнительных сегментов данных их адреса требуется указать явно с помощью специальных префиксов переопределения сегментов в команде. Адреса дополнительных сегментов данных должны содержаться в регистрах ES, GS, FS (extenSIon data segment registers).

2. Регистры управления и состояния

Микропроцессор содержит несколько регистров, которые содержат информацию о состоянии, как самого микропроцессора, так и программы, команды которой в данный момент загружены в конвейер. Это:

- регистр указателя команд EIP/IP;

- регистр флагов EFLAGS/FLAGS.

Используя эти регистры, можно получать информацию о результатах выполнения команд и влиять на состояние самого микропроцессора.

EIP/IP (instruction poINTer register) –*указатель команд*. Регистр EIP/IP имеет разрядность 32 или 16 бит и содержит смещение следующей выполняемой команды относительно содержимого сегментного регистра CS в текущем сегменте команд. Этот регистр непосредственно недоступен, но изменение его производится командами перехода.

EFLAGS/FLAGS (Flag register) – *регистр флагов*. Разрядность 32/16 бит. Отдельные биты данного регистра имеют определенное функциональное назначение и называются флагами. Флаг - это бит, принимающий значение 1 ("флаг установлен"), если выполнено некоторое условие, и значение 0 ("флаг сброшен") в противном случае. Младшая часть этого регистра полностью аналогична регистру FLAGS для i8086.

1.1.3 Регистр флагов

Регистр флагов является 32-разрядным, имеет имя EFLAGS (рис.1). Отдельные биты регистра имеют определенное функциональное назначение и называются флагами. Каждому из них присвоено определенное имя (ZF, CF и т.д). Младшие 16 бит EFLAGS представляют 16-разрядный регистр флагов FLAGS, используемый при выполнении программ, написанных для микропроцессора i086 и i286.

31	.	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

		AC	V	RF		NT		IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF	

Рис.1 Регистр флагов

Некоторые флаги принято называть флагами условий; они автоматически меняются при выполнении команд и фиксируют те или иные свойства их результата (например, равен ли он нулю). Другие флаги называются флагами состояний; они меняются из программы и оказывают влияние на дальнейшее поведение процессора (например, блокируют прерывания).

Флаги условий:

CF (carry flag) - *флаг переноса*. Принимает значение 1, если при сложении целых чисел появилась единица переноса, не "влезающая" в разрядную сетку, или если при вычитании чисел без знака первое из них было меньше второго. В командах сдвига в CF заносится бит, вышедший за разрядную сетку. CF фиксирует также особенности команды умножения.

OF (overflow flag) - *флаг переполнения*. Устанавливается в 1, если при сложении или вычитании целых чисел со знаком получился результат, по модулю превосходящий допустимую величину (произошло переполнение мантиссы и она "залезла" в знаковый разряд).

ZF (zero flag) - *флаг нуля*. Устанавливается в 1, если результат команды оказался равным 0.

SF (SIgn flag) - *флаг знака*. Устанавливается в 1, если в операции над знаковыми числами получился отрицательный результат.

PF (parity flag) - *флаг четности*. Равен 1, если результат очередной команды содержит четное количество двоичных единиц. Учитывается обычно только при операциях ввода-вывода.

AF (auxiliary carry flag) - *флаг дополнительного переноса*. Фиксирует особенности выполнения операций над двоично-десятичными числами.

Флаги состояний:

DF (direction flag) - *флаг направления*. Устанавливает направление просмотра строк в строковых командах: при DF=0 строки просматриваются "вперед" (от начала к концу), при DF=1 - в обратном направлении.

IOPL (input/output privilege level) – *уровень привилегий ввода-вывода.* Используется в защищенном режиме работы микропроцессора, для контроля доступа к командам ввода-вывода, в зависимости от привилегированности задачи.

NT (nested task) – *флаг вложенности задачи.* Используется в защищенном режиме работы микропроцессора для фиксации того факта, что одна задача вложена в другую.

Системные флаг:

IF (INTerrupt flag) - *флаг прерываний*. При IF=0 процессор перестает реагировать на поступающие к нему прерывания, при IF=1 блокировка прерываний снимается.

TF (trap flag) - *флаг трассировки*. При TF=1 после выполнения каждой команды процессор делает прерывание (с номером 1), чем можно воспользоваться при отладке программы для ее трассировки.

RF (resume flag) – *флаг возобновления*. Используется при обработке прерываний от регистров отладки.

VM (virtuAL 8086 mode) – *флаг виртуального 8086.* 1-процессор работает в режиме виртуального 8086. 0- процессор работает в реальном или защищенном режиме.

AC (ALignment check) –*флаг контроля выравнивания.* Предназначен для разрешения контроля выравнивания при обращении к памяти.

Организация памяти.

Физическая память, к которой микропроцессор имеет доступ, называется *оперативной памятью (*или оперативным запоминающим устройством - *ОЗУ).* ОЗУ представляет собой цепочку байтов, имеющих свой уникальный адрес (его номер), называемый *физическим.* Диапазон значений физических адресов от 0 до 4 Гбайт. Механизм управления памятью полностью аппаратный.

Микропроцессор аппаратно поддерживает несколько моделей использования оперативной памяти:

- *сегментированную модель*. В этой модели память для программ делится на непрерывные области памяти (сегменты), а сама программа может обращаться только к данным, которые находятся в этих сегментах;
- *страничную модель*. В этом случае оперативная память рассматривается как совокупность блоков фиксированного размера 4 Кбайта. Основное применение этой модели связано с организацией виртуальной памяти, что позволяет использовать для работы программ пространство памяти большее, чем объем физической памяти. Для микропроцессора Pentium размер возможной виртуальной памяти может достигать 4 Тбайта.

Использование и реализация этих моделей зависит от режима работы микропроцессора:

1. *Режим реальных адресов (реальный режим).* Режим аналогичный работе i8086 процессора. Необходим для функционирования программ, разработанных для ранних моделей процессоров.
2. *Защищенный режим.* В защищенном режиме появляется возможность многозадачной обработки информации, защиты памяти с помощью четырехуровнего механизма привилегий и ее страничной организации.
3. *Режим виртуального 8086.* В этом режиме появляется возможность работы нескольких программ для i8086. При этом возможна работа программ реального режима.

Далее рассмотрим только особенности работы с оперативной памятью для реального режима, в котором поддерживается только сегментированная модель организации памяти.

Сегментация – механизм адресации, обеспечивающий существование нескольких независимых адресных пространств. Сегмент представляет собой независимый, поддерживаемый на аппаратном уровне блок памяти.

Каждая программа в общем случае может состоять из любого количества сегментов, но непосредственный доступ она имеет к трем основным: кода, данных и стека – и от одного до трех дополнительных сегментов данных. Операционная система размещает сегменты программы в оперативной памяти по определенным физическим адресам, после чего помещает значения этих адресов в соответствующие регистры. Внутри сегмента программа обращается к адресам относительно начала сегмента линейно, то есть начиная с адреса 0 и заканчивая адресом, равным размеру сегмента. Относительный адрес или *смещение,* который микропроцессор использует для доступа к данным внутри сегмента, называется *эффективным.*

Формирование физического адреса в реальном режиме

В реальном режиме диапазон изменения физического адреса от 0 до 1 Мбайт. Максимальный размер сегмента 64 Кбайт. При обращении к конкретному *физическому адресу* оперативной памяти определяется адрес начала сегмента и смещение внутри сегмента. Адрес начала сегмента берется из соответствующего сегментного регистра. При этом в сегментном регистре содержатся только старшие 16 бит физического адреса начала сегмента. Недостающие младшие четыре бита 20-битного адреса получаются сдвигом значения сегментного регистра влево на 4 разряда. Операция сдвига выполняется аппаратно. Полученное 20-битное значение и является настоящим физическим адресом, соответствующим началу сегмента. То есть *физический адрес* задается как пара "сегмент:смещение", где "сегмент" (segment) - это первые16 битов начального адреса сегмента памяти, которому принадлежит ячейка, а "смещение" - 16-битовый адрес этой ячейки, отсчитанный от начала данного сегмента памяти (величина 16*сегмент+смещение дает абсолютный адрес ячейки). Если, например, в регистре CS хранится величина 1234h, тогда адресная пара 1234h:507h определяет абсолютный адрес, равный 16*1234h+507h =12340h+507h = 12847h. Такая пара записывается в виде двойного слова, причем (как и для чисел) в "перевернутом" виде: в первом слове размещается смещение, а во втором - сегмент, причем каждое из этих слов в свою очередь представлено в "перевернутом" виде. Например, пара 1234h:5678h будет записана так: | 78 | 56| 34 | 12|.

Данный механизм образования физического адреса позволяет сделать программное обеспечение перемещаемым, то есть не зависящим от конкретных адресов загрузки его в оперативной памяти.

1.3. Представление данных

Здесь и далее рассматривается машинное представление целых чисел, строк и адресов в *реальном режиме*. Представление двоично-десятичных чисел, используемых достаточно редко, не рассматривается. Что касается вещественных чисел, то в ПК нет команд вещественной арифметики (операции над этими числами реализуются программным путем или выполняются сопроцессором) и потому нет стандартного представления вещественных чисел. Кроме того, рассматриваются некоторые особенности выполнения арифметических операций.

Шестнадцатиричные числа записываются с буквой h на конце, двоичные числа - с буквой b.

1.3.1 Типы данных

В общем случае под целое число можно отвести любое число байтов, однако, система команд ПК поддерживает только числа размером в *байт, слово, двойное слово* и *учетверенное слово*.

Байт – восемь последовательно расположенных битов, пронумерованных от 0 до 7, при этом бит 0 является самым младшим значащим битом.

Слово – последовательность из двух байт, имеющих последовательные адреса. Микропроцессоры Intel имеют важную особенность – младший байт всегда хранится по меньшему адресу. *Адресом слова* считается адрес его младшего байта. Адрес старшего байта может быть использован для доступа к старшей половине слова.

Двойное слово – последовательность из четырех байтов (32 бита), расположенных по последовательным адресам. *Адресом двойного слова* считается адрес его младшего слова.

Учетверенное слово – последовательность из восьми байт (64 бита), расположенных по последовательным адресам.

В микропроцессор делается различие между целыми числами без знака (неотрицательными) и со знаком. Это объясняется тем, что в ячейках одного и того же размера можно представить больший диапазон беззнаковых чисел, чем неотрицательных знаковых чисел, и если известно заранее, что некоторая числовая величина является неотрицательной, то выгоднее рассматривать ее как беззнаковую, чем как знаковую.

Целые числа без знака - могут быть представлены в виде байта, слова или двойного слова - в зависимости от их размера. В виде байта представляются целые от 0 до 255 ($=2^8$-1), в виде слова - целые от 0 до 65535 ($=2^{16}$-1), в виде двойного слова - целые от 0 до 4 294 967 295 ($=2^{32}$-1). Числа записываются в двоичной системе счисления, занимая все разряды ячейки. Например, число 130 записывается в виде байта 10000010b (82h). Числа размером в слово хранятся в памяти в "перевернутом" виде: младшие (правые) 8 битов числа размещаются в первом байте слова, а старшие 8 битов - во втором байте (в 16-ричной системе: две правые цифры - в первом байте, две левые цифры - во втором байте). Например, число 130 (=0082h) в виде слова хранится в памяти так: | 82 | 00 |. Отметим, однако, что в регистрах числа хранятся в нормальном виде: AX | 00 82 |

"Перевернутое" представление используется и при хранении в памяти целых чисел размером в двойное слово: в первом его байте размещаются младшие 8 битов числа, во втором байте - предыдущие 8 битов и т.д. Например, число 12345678h хранится в памяти так: | 78 | 56 | 34 | 12 |.

Другими словами, в первом слове двойного слова размещаются младшие (правые) 16 битов числа, а во втором слове - старшие 16 битов, причем в каждом из этих двух слов в свою очередь используется "перевернутое" представление.

Конечно, "перевернутое" представление неудобно для людей, однако при использовании языка ассемблера это неудобство не чувствуется.

Целые числа со знаком - также представляются в виде байта, слова и двойного слова. В виде байта записываются числа от -128 до 127, в виде слова -числа от -32768 до 32767, а в виде двойного слова - числа от -2147483648 до 2147483647. При этом числа записываются в дополнительном коде: неотрицательное число записывается так же, как и беззнаковое число (т.е. в прямом коде), а отрицательное число -x (x>0) представляется беззнаковым числом 2^8-x (для байтов), 2^{16}-x (для слов) или 2^{32}-x (для двойных слов). Например, дополнительным кодом числа -6 является байт FAH (=256-6), слово FFFAH или двойное слово FFFFFFFAH. При этом байт 10000000b (=80h) трактуется как -128, а не как +128 (слово 8000h понимается как -32678), поэтому левый бит дополнительного кода всегда играет роль знакового: для неотрицательных чисел он равен 0, для отрицательных - 1.

Знаковые числа размером в слово и двойное слово записываются в памяти в "перевернутом" виде (при этом знаковый бит оказывается в последнем байте ячейки).

Иногда число-байт необходимо расширить до слова, т.е. нужно получить такое же по величине число, но размером в слово. Существует два способа такого расширения - без знака и со знаком. В любом случае исходное число-байт попадает во второй (до "переворачивания") байт слова, а вот первый байт заполняется по-разному: при расширении без знака в него записываются нулевые биты (12h -> 0012h), а при расширении со знаком в первый байт записываются нули, если число-байт было неотрицательным, и записывается восемь двоичных единиц в противном случае (81h -> FF81h). Другими словами, при расширении со знаком в первом байте слова копируется знаковый разряд числа-байта.

Аналогично происходит расширение числа-слова до двойного слова.

Неупакованное двоично-десятичное число – байтовое представление десятичной цифры от 0 до 9. Неупакованные десятичные числа хранятся как байтовые значения без знака по одной цифре в каждом байте.

Упакованное двоично-десятичное число – представляет собой упакованное представление двух десятичных цифр от 0 до 9 в одном байте. Каждая цифра хранится в своем полубайте.

1.3.2 Представление символов и строк

На символ отводится один байт памяти, в который записывается код символа - целое от 0 до 255. В ПК используется система кодировки ASCII (American Standard Code for Information INTerchange). Она, естественно, не содержит кодов русских букв, поэтому в нашей стране применяется некоторый вариант этой системы с русскими буквами (обычно это альтернативная кодировка ГОСТа).

Некоторые особенности этих систем кодировки:

- код пробела меньше кода любой буквы, цифры и вообще любого графически представимого символа;

- коды цифр упорядочены по величине цифр и не содержат пропусков, т.е. из неравенства код('0')<=код(c)<=код('9') следует, что c - цифра;

- коды больших латинских букв упорядочены согласно алфавиту и не содержат пропусков; аналогично с малыми латинскими буквами;

- (в альтернативной кодировке ГОСТа) коды русских букв (как больших, так и малых) упорядочены согласно алфавиту, но между ними есть коды других символов. Строка (последовательность символов) размещается в соседних байтах памяти (в не перевернутом виде): код первого символа строки записывается в первом байте, код второго символа - во втором байте и т.п. Адресом строки считается адрес ее первого байта.

В ПК 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.

2. Операторы программы на ассемблере

Программа на ассемблере представляет собой последовательность *операторов,* описывающих выполняемые операции. Оператором (строкой) исходной программы может быть или команда, или псевдооператор языка ассемблер.

Команды языка ассемблера представляют собой краткую нотацию системы команд микропроцессора. Команды в процессе трансляции преобразуются в машинные коды. В отличие от них *псевдооператоры* представляют собой инструкции для компилятора ассемблера и в машинные коды не преобразуются.

2.1 Команды языка ассемблера

Каждая команда языка ассемблера в исходной программе может иметь до четырех *полей* следующего вида:

[Метка:] Мнемокод [Операнд] [;Комментарий] .

Поле операнда заполняется только для тех команд, которым требуется операнд. Квадратные скобки показывают, что эти поля не обязательны. Можно набирать содержимое поля в любом месте строки, но обязательно разделять поля хотя бы одним пробелом.

Поле метки служит для присвоения *имени* команде языка ассемблер. По нему на эту команду могут ссылаться другие команды программы. Метка содержит до 31 символа и должна заканчиваться двоеточием (:). Метку можно начинать с любого символа, кроме цифры. Не используются как метки служебные слова.

Поле мнемокода содержит имя команды микропроцессора. Имена состоят из двух – шести букв. Для трансляции каждого мнемокода программы в его числовой эквивалент ассемблер использует внутреннюю таблицу.

Во многих командах, кроме мнемокода, надо указать один или два *операнда.* Ассемблер по мнемокоду узнает, сколько должно быть операндов и какого типа, а затем обрабатывает поле операндов.

В командах с двумя операндами первый из них представляет собой *приемник*, а второй – *источник*. При исполнении команды операнд-источник никогда не изменяется, в то время как операнд-приемник изменяется почти всегда.

На *поле комментариев* указывает точка с запятой (;), которая должна быть отдалена от предыдущего поля по крайне мере одним пробелом. Ассемблер игнорирует комментарии при трансляции, но сохраняет их в листинге программы.

Команды, записанные на ассемблере, преобразуется транслятором в *машинные команды* или *объектный код*.

Машинные команды занимают от 1 до 6 байтов. Код операции (КОП) занимает один или два первых байта команды. В ПК столь много различных операций, что для них не хватает 256 различных КОПов, которые можно представить в одном байте. Поэтому некоторые операции объединяются в группу, и им дается один и тот же КОП, во втором же байте этот КОП уточняется. Кроме того, во втором байте указываются типы и способ адресации операндов. Остальные байты команды указывают на операнды. У большинства команд - один или два операнда. Размер операндов - байт или слово (редко -двойное слово). Операнд может быть указан в самой команде (это т.н. непосредственный операнд), либо может находиться в одном из регистров 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, берется первый операнд. Например, большинство команд с двумя операндами реализуют действие op1 := op1 op2, где op1 - регистр или ячейка, а op2 - непосредственный операнд, регистр или ячейка.

Адрес операнда разрешено модифицировать по одному или двум регистрам. В первом случае в качестве регистра-модификатора разрешено использовать регистр BX, BP, SI или DI (и никакой иной). 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.

В ассемблере адреса в командах записываются в виде одной из следующих конструкции:

A, A[M] или A[M1][M2],

где A - адрес, M - регистр BX, BP, SI или DI, M1 - регистр BX или BP, а M2 - регистр SI или DI. Во втором и третьем варианте A может отсутствовать, в этом случае считается, что A=0.

При выполнении команды процессор прежде всего вычисляет т.н. *эффективный* (исполнительный) адрес - как сумму адреса, заданного в команде, и текущих значений указанных регистров-модификаторов, причем все эти величины рассматриваются как неотрицательные и суммирование ведется по модулю 2^{16} ([r] означает содержимое регистра r):

A : Аисп = A

A[M] : Аисп = A+[M] (mod 2^{16})

A[M1][M2]: Аисп = A+[M1]+[M2] (mod 2^{16})

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

2.2. Режимы адресации и форматы машинных команд

Микропроцессор Intel предоставляет множество способов доступа к операндам. Операнды могут содержаться в регистрах, самих командах, в памяти или в портах ввода-вывод. Режимы адресации разделяются на семь групп:

1. Регистровая адресация.
2. Непосредственная адресация.
3. Прямая адресация.
4. Косвенная регистровая адресация.
5. Адресация по базе.
6. Прямая адресация с индексированием.
7. Адресация по базе с индексированием.

Микропроцессор выбирает режим адресации по значению *поля режима* команды. Ассемблер присваивает то или иное значение *полю режима* в зависимости от того, какой вид имеют операнды в исходной программе. Например, если есть команда

MOV AX,BX

То ассемблер закодирует оба операнда (AX, BX) для регистровой адресации. Если же поместить регистр BX в квадратные скобки:

MOV AX,[BX]

То Ассемблер закодирует операнд-источник для косвенной регистровой адресации.

В табл. 1 приведены форматы операндов языка ассемблер и указан какой из регистров сегмента используется для вычисления физического адреса.

Табл. 1

Режим адресации	Формат операнда	Регистр сегмента
Регистровый	Регистр	Не используется
Непосредственный	Данное	Не используется
Прямой	Метка Сдвиг	DS DS
Косвенный регистровый	[BX]	DS

	[BP] [DI] [SI]	SS DS DS
По базе	[BX]+сдвиг [BP]+сдвиг	DS CS
Прямой с индексированием	[DI]+сдвиг [SI]+сдвиг	DS DS
По базе с индексированием	[BX][SI]+сдвиг [BX][DI]+сдвиг [BP][SI]+сдвиг [BP][DI]+сдвиг	DS DS SS SS

В зависимости от формата операнда и режима адресации формируется *объектный код* или машинная команда. Форматы машинных команд достаточно разнообразны. Приведем лишь основные форматы команд с двумя операндами.

1) Формат "регистр-регистр" (2байта):

КОП d w 11 reg1 reg2

7 ... 2 1 0 7 6 5 4 3 2 1 0

Команды этого формата описывают обычно действие reg1:=reg1 reg2 или reg2:=reg2 reg1. Поле КОП первого байта указывает на операцию ( ), которую надо выполнить. Бит w определяет размер операндов, а бит d указывает, в какой из регистров записывается результат:

w = 1 - слова d = 1 - reg1:=reg1 reg2

= 0 - байты = 0 - reg2:=reg2 reg1

Во втором байте два левых бита фиксированы (для данного формата), а трехбитовые поля reg1 и reg2 указывают на регистры, участвующие в операции, согласно следующей таблице:

Табл.2

reg	W=1	W=0	Reg	W=1	W=0
000	AX	AL	100	SP	AH

001	CX	CL	101	BP	CH
010	DX	DL	110	SI	DH
011	BX	BL	111	DI	BH

2) Формат "регистр-память" (2-4 байта):

КОП |d|w| |mod|reg|mem| |адрес (0-2 байта)

Эти команды описывают операции reg:=reg mem или mem:=mem reg. Бит w первого байта определяет размер операндов (см. выше), а бит d указывает, куда записывается результат: в регистр (d=1) или в ячейку памяти (d=0). Трехбитовое поле reg второго байта указывает операнд-регистр (см. выше), двухбитовое поле mod определяет, сколько байтов в команде занимает операнд-адрес (00 - 0 байтов, 01 - 1 байт, 10 - 2 байта), а трехбитовое поле mem указывает способ модификации этого адреса. В следующей таблице указаны правила вычисления исполнительного адреса в зависимости от значений полей mod и mem (a8 - адрес размером в байт, a16 - адрес размером в слово):

Табл.3

mem mod 00 01 10

000 [BX]+[SI] [BX]+[SI]+a8 [BX]+[SI]+a16

001 [BX]+[DI] [BX]+[DI]+a8 [BX]+[DI]+a16

010 [BP]+[SI] [BP]+[SI]+a8 [BP]+[SI]+a16

011 [BP]+[DI] [BP]+[DI]+a8 [BP]+[DI]+a16

100 [SI] [SI]+a8 [SI]+a16

101 [DI] [DI]+a8 [DI]+a16

110 a16 [BP]+a8 [BP]+a16

111 [BX] [BX]+a8 [BX]+a16

Замечания. Если в команде не задан адрес, то он считается нулевым.

Если адрес задан в виде байта (a8), то он автоматически расширяется со знаком до слова (a16). Случай mod=00 и mem=110 указывает на отсутствие регистров-модификаторов, при этом адрес должет иметь размер слова (адресное выражение [BP] ассемблер транслирует в mod=01 и mem=110 при a8=0). Случай mod=11 соответствует формату "регистр-регистр".

3) Формат "регистр-непосредственный операнд" (3-4 байта):

КОП |s|w| |11|КОП"|reg|

|непосред.операнд (1-2 б)

Команды этого формата описывают операции reg:=reg immed (immed - непосредственный операнд). Бит w указывает на размер операндов, а поле reg - на регистр-операнд (см. выше). Поле КОП в первом байте определяет лишь класс операции (например, класс сложения), уточняет же операцию поле КОП" из второго байта. Непосредственный операнд может занимать 1 или 2 байта - в зависимости от значения бита w, при этом операнд-слово записывается в команде в "перевернутом" виде. Ради экономии памяти в ПК предусмотрен случай, когда в операции над словами непосредственный операнд может быть задан байтом (на этот случай указывает 1 в бите s при w=1), и тогда перед выполнением операции байт автоматически расширяется (со знаком) до слова.

4) Формат "память-непосредственный операнд" (3-6 байтов):

КОП |s|w|

|mod|КОП"|mem| |адрес (0-2б)| |непоср.оп (1-2б)|

Команды этого формата описывают операции типа mem:=mem immed. Смысл всех полей - тот же, что и в предыдущих форматах.

Помимо рассмотренных в ПК используются и другие форматы команды с двумя операндами; так, предусмотрен специальный формат для команд, один из операндов которых фиксирован (обычно это регистр AX). Имеют свои форматы и команды с другим числом операндов.

Из сказанного ясно, что одна и та же операция в зависимости от типов операндов записывается в виде различных машинных команд: например, в ПК имеется 28 команд пересылки байтов и слов. В то же время в ассемблере все эти "родственные" команды записываются единообразно: например, все команды пересылки имеют одну и ту же символьную форму записи:

MOV op1,op2 (op1:=op2)

Анализируя типы операндов, ассемблер сам выбирает подходящую машинную команду.

Регистры указываются своими именами, например:

MOV AX,SI ;оба операнда - регистры

Непосредственные операнды задаются константными выражениями (их значениями являются константы-числа), например:

MOV BH,5 ;5 - непосредственный операнд

MOV DI,SIZE_X ;SIZE_X (число байтов, занимаемых переменной X) - непосредственный операнд

Адреса описываются адресными выражениями (например, именами переменных), которые могут быть модифицированы по одному или двум регистрам; например, в следующих командах первые операнды задают адреса:

```
MOV X,AH

MOV X[BX][DI],5

MOV [BX],CL
```

При записи команд в символьной форме необходимо внимательно следить за правильным указанием типа (размера) операндов, чтобы не было ошибок. Тип обычно определяется по внешнему виду одного из них, например:

```
MOV AH,5 ;пересылка байта, т.к. AH - байтовый регистр

MOV AX,5 ;пересылка слова, т.к. AX - 16-битовый регистр;(операнд 5 может быть байтом
и словом, по нему ;нельзя определить размер пересылаемой величины)

MOV [BX],300 ;пересылка слова, т.к. число 300 не может быть байтом.
```

Если по внешнему виду можно однозначно определить тип обоих операндов, тогда эти типы должны совпадать, иначе ассемблер зафиксирует ошибку. Примеры:

```
MOV DS,AX ;оба операнда имеют размер слова

MOV CX,BH ;ошибка: регистры CX и BH имеют разные размеры

MOV DL,300 ;ошибка: DL - байтовый регистр, а число 300 не

;может быть байтом
```

Возможны ситуации, когда по внешнему виду операндов нельзя определить тип ни одного из них, как, например, в команде

```
MOV [BX],5
```

Здесь число 5 может быть и байтом, и словом, а адрес из регистра BX может указывать и на байт памяти, и на слово. В подобных ситуациях ассемблер фиксирует ошибку. Чтобы избежать ее, надо уточнить тип одного из операндов с помощью оператора с названием PTR:

```
MOV BYTE PTR [BX],5 ;пересылка байта

MOV WORD PTR [BX],5 ;пересылка слова
```

(Операторы - это разновидность выражений языка ассемблер, аналогичные функциям.)

Оператор PTR необходим и в том случае, когда надо изменить тип, предписанный имени при его описании. Если, например, X описано как имя переменной размером в слово:

X DW 999

и если надо записать в байтовый регистр AH значение только первого байта этого слова, тогда воспользоваться командой

MOV AH,X

нельзя, т.к. ее операнды имеют разный размер. Эту команду следует записать несколько иначе:

MOV AH,BYTE PTR X

Здесь конструкция BYTE PTR X означает адрес X, но уже рассматриваемый не как адрес слова, а как адрес байта. (Напомним, что с одного и того же адреса может начинаться байт, слово и двойное слово; оператор PTR уточняет, ячейку какого размера мы имеем в виду.)

3. Псевдооператоры

Псевдооператоры управляют работой Ассемблера, а не микропроцессора. С помощью псевдоопереторов можно определять сегменты и процедуры, давать имена командам и элементам данных, резервировать рабочие области памяти, управлять листингом программы и другие выполнять множество других функций. В отличии от команд языка ассемблера большинство псевдооператоров не генерируют объектного кода.

3.1 Директивы определения данных

Для того чтобы в программе на ассемблере зарезервировать ячейки памяти под константы и переменные, необходимо воспользоваться директивами определения данных - с названиями DB (описывает данные размером в байт), DW (размером в слово) и DD (размером в двойное слово). (Директивы, или команды ассемблеру, - это предложения программы, которыми ее автор сообщает какую-то информацию ассемблеру или просит что-то сделать дополнительно, помимо перевода символьных команд на машинный язык.)

В простейшем случае в директиве DB, DW или DD описывается одна константа, которой дается имя для последующих ссылок на нее. По этой директиве ассемблер формирует машинное представление константы (в частности, если надо, "переворачивает" ее) 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 ассемблере числа записываются в нормальном (неперевернутом) виде в системах счисления с основанием 10, 16, 8 или 2. Десятичные числа записываются как обычно, за шестнадцатиричным числом ставится буква h (если число начинается с "цифры" A, B, ..., F, то вначале обязателен 0), за восьмиричным числом - буква q или o, за двоичным числом - буква b.

Примеры:

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 ;-1 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); в последнем случае ассемблер сам заменит символ на его код. Например, следующие директивы эквивалентны (2A - код звездочки в ASCII):

CH DB 02AH

CH DB '*'

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

ADR DW CH

будет отведено слово памяти, которому дается имя ADR и в которое запишется адрес (смещение), соответствующий имени CH. Если такое же имя описать в директиве DD, то ассемблер автоматически добавит к смещению имени его сегмент и запишет смещение в первую половину двойного слова, а сегмент - во вторую половину.

По любой из директив DB, DW и DD можно описать переменную, т.е. отвести ячейку, не дав ей начального значения. В этом случае в правой части директивы указывается вопросительный знак:

F DW ? ;отвести слово и дать ему имя F, ничего в этот байт не записывать

В одной директиве можно описать сразу несколько констант и/или переменных одного и того же размера, для чего их надо перечислить через запятую. Они размещаются в соседних ячейках памяти. Пример:

G DB 200, -5, 10h, ?, 'F'

Имя, указанное в директиве, считается именующим первую из констант. Для ссылок на остальные в ассемблере используются выражения вида <имя>+<целое>; например, для доступа к байту с числом -5 надо указать выражение G+1, для доступа к байту с 10h - выражение G+2 и т.д.

Если в директиве DB перечислены только символы, например:

S DB 'a','+','b'

тогда эту директиву можно записать короче, заключив все эти символы в одни кавычки:

S DB 'a+b'

И, наконец, если в директиве описывается несколько одинаковых констант (переменных), то можно воспользоваться конструкцией повторения

K DUP(a,b,...,c)

которая эквивалентна повторенной k раз последовательности a,b,...,c.

Например, директивы

V1 DB 0,0,0,0,0

V2 DW ?,?,?,?,?,?,?,?,?,'a',1,2,1,2,1,2,1,2

можно записать более коротко таким образом:

V1 DB 5 DUP(0)

V2 DW 9 DUP(?), 'a', 4 DUP(1,2)

3.2 Структура программы на ассемблере

Рассмотрим, как правильно оформлять последовательность команд, чтобы транслятор мог их обработать, а микропроцессор выполнить.

3.2.1 Программные сегменты. Директива ASSUME

Для того чтобы указать, что некоторая группа предложений программы на ассемблере образуют единый сегмент памяти, они оформляются как программный сегмент: перед ними ставится директива SEGMENT с операндами, после 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 последовательность таких программных сегментов, в конце которой указывается директива конца программы END, например:

DT1 SEGMENT PARA PUBLIC ‘DATA’ ; сегмент данных с именем DT1

A DB 0

B DW ?

DT1 ENDS

CODE SEGMENT ; кодовый сегмент CODE

ASSUME CS:CODE, DS:DT1

MAIN PROC

MOV AX,DT1 ;инициализация сегмента

MOV DS,AX ;данных

```
MOV AX,A

...

MOV AX,4C00H ; выход из

INT 21H ; программы

MAIN ENDP ; конец процедуры MAIN

CODE ENDS ; конец сегмента кода

END MAIN ;конец программы
```

Важно отметить, что функциональное назначение сегмента несколько шире, чем просто разбиение программ на блоки кода, данных и стека. Сегментация является частью более общего механизма, связанного с концепцией модульного программирования, она предполагает унификацию оформления объектных модулей, создаваемых компилятором, в том числе с разных языков программирования. Это позволяет объединять программы, написанные на разных языках. Именно для реализации различных вариантов такого объединения и предназначены операнды в директиве SEGMENT:

SEGMENT <тип выравнивания><тип комбинирования><класс><тип размера сегмента>

Первый операнд в директиве – *атрибут выравнивания сегмента* (тип выравнивания) сообщает компановщику о том, что нужно обеспечить размещение начала сегмента на заданной границе. Допустимые значения атрибута:

BYTE – выравнивание не выполняется. Сегмент может начинаться с любого адреса.

WORD – сегмент начинается по адресу, кратному двум.

DWORD – сегмент начинается по адресу кратному четырем.

PARA - сегмент начинается по адресу, кратному 16. Принимается по умолчанию.

PAGE -сегмент начинается по адресу кратному 256.

NEMPAGE – сегмент начинается по адресу, кратному 4 Кбайт.

Следующий *атрибут комбинирования сегментов*(комбинаторный тип) сообщает компоновщику, как нужно комбинировать сегменты различных модулей, имеющих одно и то же имя. По умолчанию атрибут комбинирования принимает значение PRIVATE. Атрибуты комбинирования могут быть следующими:

PRIVATE – сегмент не будет объединяться с другими сегментами с тем же именем вне данного модуля.

PUBLIC – компоновщик соединит все сегменты с одинаковыми именами.

COMMON – располагает все сегменты с одним и тем же именем по одному адресу.

AT xxxx – располагает сегмент по абсолютному адресу параграфа. Абсолютный адрес параграфа задается выражением xxxx.

STACK – определение сегмента стека. Заставляет компоновщик соединять все одноименные сегменты и вычислять адреса этих сегментов относительно регистра SS.

Атрибут класса сегмента (тип класса) – это заключенная в кавычки строка, помогающая компоновщику определить соответствующий порядок следования сегментов при объединении программ.

Все ссылки на предложения одного программного сегмента ассемблер сегментирует по умолчанию по одному и тому же сегментному регистру. По какому именно - устанавливается специальной директивой ASSUME. В нашем примере эта директива определяет, что все ссылки на сегмент CODE должны, если явно не указан сегментный регистр, сегментироваться по регистру CS, все ссылки на DT1 - по регистру DS, а все ссылки на DT2 - по регистру ES.