Московский государственный технический университет
имени Н.Э. Баумана

Факультет «Информатика и вычислительная техника»

Кафедра «Компьютерные системы и сети»

Г.С. Иванова, Т.Н. Ничушкина

ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ АССЕМБЛЕРА IA-32 (IA-64)

Электронное учебное пособие по дисциплине

«Машинно-зависимые языки и основы компиляции»

Москва

Оглавление

УДК 004.432

Рецензент:

Г.С. Иванова, Т.Н. Ничушкина

Основы программирования на ассемблере *IA*-32 (*IA*-64). Учебное пособие по дисциплине «Машинно-зависимые языки и основы компиляции». Главы 1-2. - М.: МГТУ имени Н.Э. Баумана, 2022. 51 с.

Учебное пособие ориентировано на студентов, начинающих изучать основы программирования на 32-х или 64-х разрядных ассемблерах с использованием транслятора *NASM*. Оно содержит необходимые сведения об архитектуре процессоров фирмы *Intel*, начиная с родоначальника ряда – процессора *i*8086, регистрах, входящих в эти процессоры, схемах адресации оперативной памяти, а также описание структуры и формата машинных команд, без знания которых многие особенности программирования на ассемблере остаются непонятными.

Очень важно включение в учебное пособие информации о различных видах синтаксиса ассемблера. Также пособие включает сведения о структуре программы на ассемблере и основных директивах транслятора *NASM*. Приведено описание форматов основных машинных команд ассемблера и правилах их записи. Кроме этого в пособии обсуждаются некоторые приемы программирования на ассемблере, такие как программирование ветвлений, организация циклов разного вида и обработки массивов и матриц.

Для студентов МГТУ имени Н.Э. Баумана, обучающихся по программе бакалавриата направлений 09.03.01 «Информатика и вычислительная техника» и 09.03.03 «Прикладная информатика».

Рекомендовано Учебно-методической комиссией НУК «Информатика и системы управления» МГТУ им. Н.Э. Баумана

Электронное учебное издание

Иванова Галина Сергеевна
Ничушкина Татьяна Николаевна

Основы программирования на ассемблере *IA*-32 (*IA*-64)

Учебное пособие
по дисциплине Машинно-зависимые языки и основы компиляции

Оглавление

Оглавление

ВВЕДЕНИЕ

Язык ассемблера или ассемблер (англ. *assembly language*) — язык низкого уровня с командами, обычно соответствующими командам процессора (так называемым машинным командам). Это соответствие позволяет отнести язык к группе машинно-зависимых, к которой относятся также машинные языки.

В сответствии с этим для изучения основ программирования на языке ассемблера прежде всего необходимо иметь представление о структуре современных процессоров типа *Intel*, структуре машинных команд и принципах адресации оперативной памяти.

Далее в пособии рассмотрены основные машинные команды ассемблера и принципы реализации структурных конструкций алгоритмов, таких как ветвления и циклы. Также рассмотрены особенности работы с командами обработки строк и приведены примеры использования этих команд для реализации обработки символьной информации.

Изучение основ программирования на языке ассемблера является необходимой частью обучения специалистов в области проектирования, как аппаратных, так и программных средств вычислительной технике, поскольку позволяет отчетливо понимать, как работает вычислительная машина в процессе выполнения программы.

При выполнении лабораторных работ по дисциплине используется следующее программное обеспечение, работающее под управлением операционной системы *Linux Astra*:

- программа эмуляции терминала *Fly*;
- текстовый редактор *Kate*;
- шестнадцатиричный текстовый редактор *GHex*;
- транслятор ассемблера *Nasm*;
- компоновщик *Ld*;
- отладчик *Edb*.

Все указанное выше программное обеспечение является свободным и при отсутствии на компьютере может быть загружено из репозитория *Astra Linux*.

1 ВЫЧИСЛИТЕЛЬНЫЕ СИСТЕМЫ НА БАЗЕ ПРОЦЕССОРОВ С АРХИТЕКТУРОЙ INTEL X86

1.1 Архитектура «с общей шиной»

Архитектурой вычислительной системы **(*ВС*)** называют совокупность основных характеристик системы, определяющих особенности ее функционирования.

С программной точки зрения архитектура процессора — это совместимость с определённым набором команд, их структурой, системой адресации, набором регистров и способом исполнения. В настоящем пособии рассматриваются процессоры с архитектурой *Intel x*86, ряд которых начинается с процессора *i*8086.

ВС с процессорами *Intel x86* строятся на базе архитектуры «с общей шиной» (рисунок 1). В этом случае процессор взаимодействует со всеми остальными устройствами через *общую или системную шину,* которая включает шины управления, адреса и данных.

Рисунок 1 – Архитектура ВС «с общей шиной»

Очевидным недостатком такой архитектуры является невозможность одновременной обработки запросов на передачу информации от разных устройств. Все запросы обрабатываются системной шиной последовательно, что может существенно замедлять об-

работку. В настоящее время этот недостаток преодолевается увеличением рабочей частоты системной шины. Также следует иметь в виду, что сегодня в основном используются многопроцессорные варианты ВС.

Однако рассмотрение особенностей построения процессоров *Intel x*86 мы начнем с родоначальника семейства – процессора *i*8086, который определил основные особенности, присущие процессорам этого ряда.

1.2 Процессор i8086

1.2.1 Структурная схема процессора

На рисунке 2 представлена упрощенная структурная схема процессора *i*8086. В его состав входят: устройство управления (УУ), арифметико-логическое устройство (АЛУ), блок преобразования (формирования) адресов и регистры.

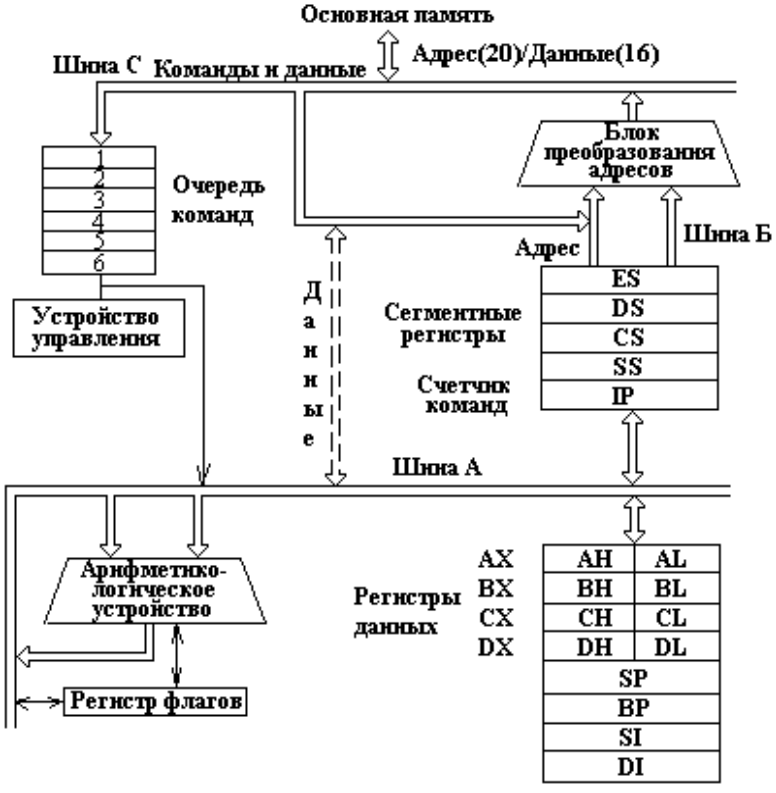


Рисунок 2 – Структура процессора *i*8086

УУ дешифрирует коды команд и формирует необходимые управляющие сигналы. АЛУ осуществляет необходимые арифметические и логические преобразования данных. В блоке преобразования адресов формируются физические адреса данных, расположенных в основной памяти. И, наконец, регистры – внутренняя память процессора – используются для хранения управляющей информации (в основном адресов и данных).

Всего в состав процессора *i*8086 входят четырнадцать 16-ти битовых регистров, каждый из которых может иметь специальное назначение, описанное далее:

а) четыре регистра общего назначения (называемые также регистрами данных):

AX – регистр-аккумулятор,

BX – базовый регистр,

CX – счетчик,

DX – регистр-расширитель аккумулятора;

б) три адресных регистра, которые должны использоваться для хранения частей адреса данных или применяется соответствующая команда:

SI – регистр индекса источника,

DI – регистр индекса приемника,

BP – регистр-указатель базы стека;

в) три управляющих регистра:

SP – регистр-указатель стека,

IP – регистр-счетчик команд,

Flags – регистр флагов;

г) четыре сегментных регистра:

CS – регистр сегмента кодов,

DS – регистр сегмента данных,

ES – регистр дополнительного сегмента данных,

SS – регистр сегмента стека.

1.2.2 Организация основной памяти вычислительной системы на базе процессоров *i*8086

Минимальной адресуемой единицей основной памяти является *байт*, состоящий из 8 битов. Память представляет собой последовательность байтов. *Номер байта является его физическим адресом в устройстве памяти.*

Адресовать отдельно бит нельзя. Если необходимо получить доступ к определенному биту, то сначала ищется соответствующий байт, а затем уже в нем – нужный бит.

Для размещения программ и данных в основной памяти выделяются специальные области – сегменты.

Сегмент при 16-ти разрядной адресации – фрагмент памяти, начинающийся с адреса кратного 16 и имеющий размер от 1 байта до 64 Кб.

Следовательно, базовый адрес сегмента всегда содержит в 4-х младших разрядах нули. Старшая часть адреса сегмента без последних четырех нулей называется *сегментным адресом* и хранится в одном из 4-х сегментных регистров. При этом каждый сегментный регистр используется для хранения адреса определенного сегмента:

- **CS** – адреса сегмента кодов, т.е. собственно программы;
- **DS**, **ES** – адреса сегмента данных;
- **SS** – адреса сегмента стека.

Физический адрес любых данных в памяти формируется из 16-ти битового смещения и 16-ти битового сегментного адреса по специальной схеме. Сначала к сегментному адресу аппаратно дописываются 4 двоичных нуля. В результате получается 20-ти битовый физический адрес начала сегмента. Затем выполняется сложение 20-ти битового базового адреса сегмента и 16-ти битового смещения. Откуда получается 20-ти битовый (20-ти разрядный) физический адрес данных (рисунок 3).

Рисунок 3 – Формирование 20-ти битового физического адреса

Таким образом для адресации основной памяти в микропроцессоре *i*8086 предусматриваются 20-битовые адреса, что позволяет работать с основной памятью до 1 Мб.

Оглавление

Если программа включает более чем один сегмент кодов, данных и стека, то сегментные регистры в процессе ее работы перегружаются – переадресуются на нужные сегменты.

Смещение для каждого типа сегмента формируется по своим правилам (рисунок 4). Для стека смещение хранится в регистре **SP**, для сегмента кодов – в **IP**, а для сегментов данных – рассчитывается в соответствие с форматом команды, как исполнительный адрес.

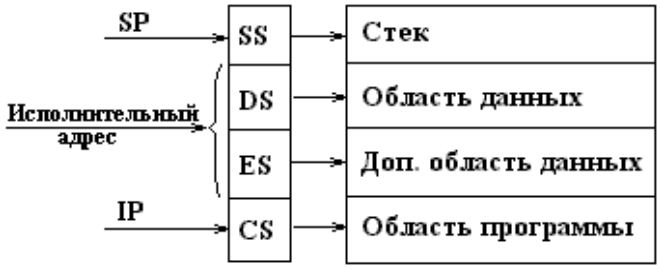


Рисунок 4 – Адресация различных типов сегментов

Стек при 16-ти разрядной адресации представляет собой специальным образом организованную область памяти, осуществляющую последовательную запись элементов данных длиной 2 байта (слово) и чтение их в порядке, обратном порядку записи. Для хранения адреса последнего слова, занесенного в стек, служит регистр-указатель стека **SP**. Каждый раз при записи данных значение **SP** уменьшается на 2, а при чтении – увеличивается на 2 (рисунок 5). Таким образом, стек растет в область младших адресов, т.е. при заполнении стека физический адрес вершины уменьшается. Соответственно в начальный момент времени указатель **SP** должен содержать максимально возможное для заданного размера стека смещение.

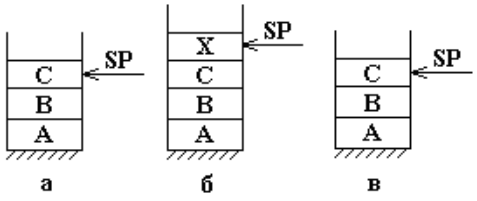


Рисунок 5 – Процессы записи в стек и чтения из стека:
а – текущее состояние стека, *б* – запись **X**, *в* – чтение **X**

Стек используется для временного хранения данных и адресов, например при вызове подпрограмм, когда в стек заносится адрес возврата и значения параметров, передаваемых в подпрограмму.

Оглавление

Формат команд процессора i8086 позволяет указывать в команде только один операнд, размещенный в основной памяти, т.е. одной командой нельзя, например, сложить содержимое двух ячеек памяти.

Принципиально допускается 8 способов задания исполнительного адреса операндов, размещенных в основной памяти:

1) <содержимое регистра **SI>** + <смещение, заданное в команде>;
2) <содержимое регистра **DI>** + <смещение, заданное в команде>;
3) <содержимое регистра **BP>** + <смещение, заданное в команде>;
4) <содержимое регистра **BX>** + <смещение, заданное в команде>;
5) <содержимое регистров **BP** + **SI>** + <смещение, заданное в команде> ;
6) <содержимое регистров **BP** + **DI>** +<смещение, заданное в команде> ;
7) <содержимое регистров **BX** + **SI>** + <смещение, заданное в команде>;
8) <содержимое регистров **BX** + **DI>** + <смещение, заданное в команде>.

Во всех случаях исполнительный адрес операнда определяется как сумма содержимого указанных регистров и смещения, заданного в команде и представляющего собой одно- или двухбайтовое число.

1.2.3 Выполнение программы

Содержимое регистров **CS** и **IP**, в которых хранятся базовый адрес сегмента кодов и смещение очередной команды относительно начала сегмента, определяет физический адрес команды, которая должна быть выполнена на следующем шаге.

По указанному адресу из основной памяти считывается команда и пересылается в процессор. Код команды длиной от 1 до 8 байт поступает в очередь команд, откуда передается в устройство управления для дешифрации.

Если при выполнении команды требуются данные, расположенные в основной памяти, то в специальном поле кода команды указывается способ адресации, согласно которому вычисляется исполнительный, а затем и физический адрес данных.

Данные, считанные из основной памяти по указанному адресу, пересылаются в регистр данных или в арифметико-логическое устройство и обрабатываются в соответствии

с кодом команды. Результат помещается в соответствии с командой либо в регистры, либо в заданную область основной памяти.

Если выполненная команда не являлась командой передачи управления, то содержимое регистра **`IP`** увеличивается на длину выполненной команды, в противном случае в регистр **`IP`** заносится исполнительный адрес следующей команды.

Затем процесс повторяется.

Флажковый регистр. На рисунке 6 представлена структура флажкового регистра **`Flags`** процессора *i*8086, в котором в виде однобитовых признаков по принципу ДА – НЕТ (ВКЛЮЧЕНО – ВЫКЛЮЧЕНО) фиксируется информация о ходе выполнения машинных команд, например арифметических.

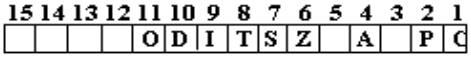


Рисунок 6 – Структура флажкового регистра

Основные флаги флажкового регистра имеют следующее назначение:

OF – флаг переполнения разрядной сетки: 1 – результат операции не умещается в отведенное под него место;

DF – направление обработки строк: 0 – от младших адресов к старшим, 1 – от старших к младшим;

IF – разрешение прерывания;

SF– признак знака: 1 – результат больше нуля, 0 – результат меньше нуля;

ZF – признак нуля: 1 – результат равен нулю;

AF – признак наличия переноса из тетрады (в настоящее время не используется);

CF – признак переноса: 1 – при выполнении операции формируется перенос (для операции сложения) или заем (для операции вычитания).

В последующем эта информация может использоваться, например, командами условной передачи управления.

1.3 Программная модель 32-х и 64-х разрядных процессоров семейства Intel x86

Структура современных процессоров семейств *IA*-32 (Intel Architecture, 32-bit) и тем более *IA*-64 (Intel Architecture, 64-bit) очень сложна, поскольку в них аппаратно реализована совокупность параллельных конвейеров для ускорения выполнения операций. На рисунке 7 современный процессор рассматриваемого семейства представлен в виде набора основных блоков. Единственно в случае 64-х разрядного процессора шины данных и адреса также 64-х разрядные.

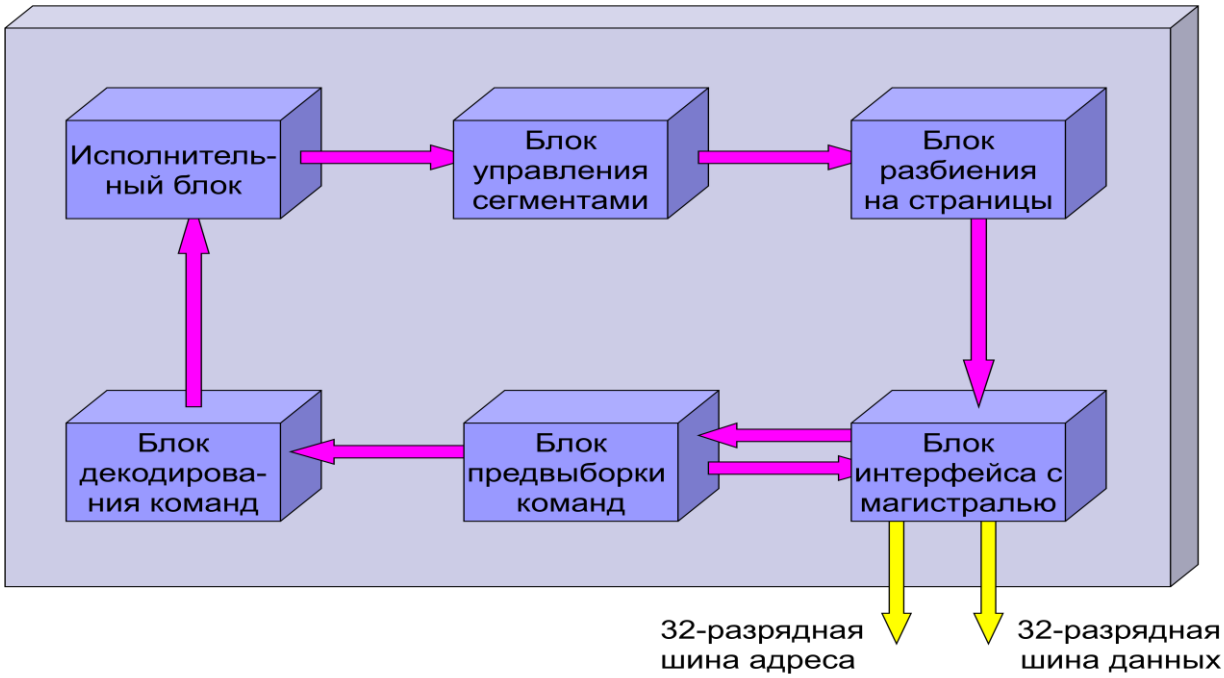


Рисунок 7 – Структура процессора семейства *IA*-32

По аналогии с процессором *i*8086 несложно определить функции основных блоков.

Блок интерфейса с магистралью управляет передачей команд и данных из памяти в процессор и результатов – обратно в оперативную память.

Блок предвыборки команд отвечает за чтение очередных команд из сегмента кодов.

Блок декодирования команд осуществляет расшифровку команды и формирование последовательности управляющих сигналов для ее выполнения (аналог УУ).

Исполнительный блок согласно названию выполняет команду (аналог АЛУ).

Блоки управления сегментами и страницами обеспечивают формирование физического адреса следующих команд и необходимых данных. При этом ограничение на нахождение не более одного операнда команды в оперативной памяти сохраняется.

1.3.1 Регистры общего назначения

Большинство регистров процессоров семейства *IA*-32 32-х разрядные. Они включают 16-ти и 8-ми разрядные регистры данных, имевшиеся в прародителе *i*8086, как младшую часть (рисунок 8), и обеспечивают доступ к ним по указанным именам.

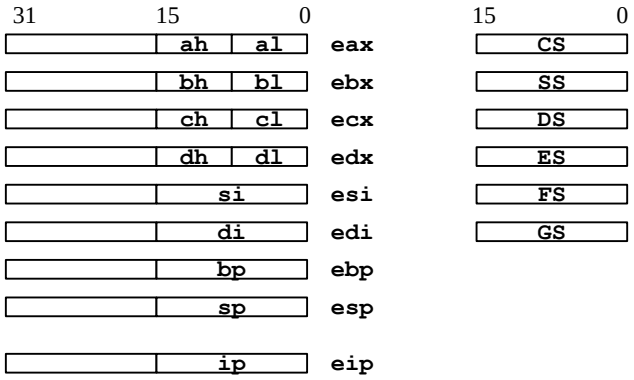


Рисунок 8 – Структура регистров данных процессоров *IA*-32

Сегментные регистры *IA*-32 остались 16-ти разрядными, но их количество увеличилось до 6. Добавленные регистры позволяют адресовать еще два сегмента данных. В защищенном режиме *IA*-32 сегментные регистры хранят не адрес сегмента, а номер (индекс) специального дескриптора, который содержит базовый адрес сегмента, его размер и атрибуты (схема адресации памяти в защищенном режиме рассмотрена далее).

Процессор *IA*-64 имеет сходное строение регистров, однако в нем добавлены 64-х разрядные регистры, имена которых начинаются с буквы «*R*». Кроме этого набор регистров данных этих процессоров расширен. В таблице 1 номера добавленных регистров выделены красным цветом.

Регистры данных процессоров *IA*-32 и *IA*-64 со сходными регистрам *i*8086 именами сохранили свое специальное назначение. Так регистры **eax** (**rax**) по-прежнему используются в арифметических командах в качестве регистра-аккумулятора, регистр **edx** (**rdx**) – в качестве его расширителя, регистр **ecx** (**rcx**) – в качестве счетчика цикла и т.п.

Таблица 1 – Регистры 64-х разрядного процессора

64-х разрядный регистр	Младшие 32 бита регистра	Младшие 16 бит регистра	Младшие 8 бит регистра
rax	**eax**	**ax**	**al**
rbx	**ebx**	**bx**	**bl**
rcx	**ecx**	**cx**	**cl**
rdx	**edx**	**dx**	**dl**
rsi	**esi**	**si**	**sil**
rdi	**edi**	**di**	**dil**
rbp	**ebp**	**bp**	**bpl**
rsp	**esp**	**sp**	**spl**
r8	**r8d**	**r8w**	**r8b**
r9	**r9d**	**r9w**	**r9b**
r10	**r10d**	**r10w**	**r10b**
r11	**r11d**	**r11w**	**r11b**
r12	**r12d**	**r12w**	**r12b**
r13	**r13d**	**r13w**	**r13b**
r14	**r14d**	**r14w**	**r14b**
r15	**r15d**	**r15w**	**r15b**

Регистр **eip** (**rip** для *IA*-64) – также сохранил свое назначение. Он используется для хранения адреса следующей команды, выполняемой процессором.

1.3.2 Режимы адресации. Схема адресации защищенного режима

Процессоры *IA*-32 и *IA*-64 поддерживают два режима адресации:

• *реальный* – в этом режиме адрес формируется аналогично *i*8086, 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;

• *защищенный* – в этом режиме используется 32-х разрядная или соответственно 64-х разрядная адресация, предусматривающие несколько вариантов защиты, откуда и появилось название этого режима.

Реальный режим – упрощенный, он используется при включении компьютера до загрузки операционной системы. Основным режимом адресации оперативной памяти является защищенный, при котором возможно использование всей мощности компьютера.

Требование сохранить возможность выполнения программ, написанных для младших процессоров типа *Intel*, привело к тому, что схемы 32-х и 64-х разрядной адресации является многокомпонентными. На рисунке 9 показана схема 32-х разрядной адресации.

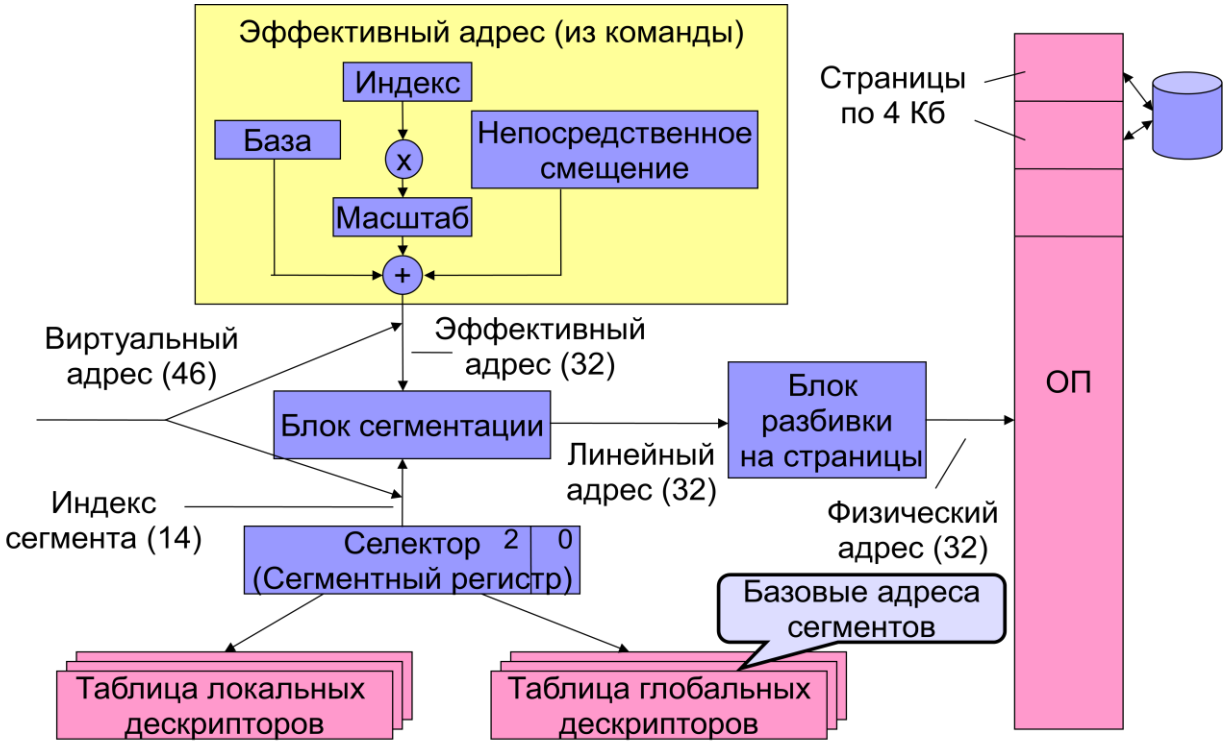


Рисунок 9 – Схема 32-х разрядной адресации в защищенном режиме

Согласно схеме 32-х разрядный эффективный или исполнительный адрес данных формируется в соответствии с информацией, указанной в явном виде в машинной команде.

При 32-х разрядной адресации по-прежнему используется сегментная организация памяти как для процессора *i*8086, но размер сегмента уже не ограничивается 64 Кб. Теоретически он может достигать 2^{32} (4 Гб) – при 32-х разрядной адресации (и 2^{64} – при 64-х разрядной адресации).

32-х разрядные адреса базы сегмента хранятся не в виде сегментного адреса в сегментном регистре, как при 16-ти разрядной адресации, а полностью в специальных внутренних регистрах процессора – *дескрипторах*. Номер дескриптора заносится в 14 бит сегментного регистра, который в этом режиме называется *селектором*. Один бит селектора из этих 14-ти отвечает за выбор таблицы локальных или глобальных дескрипторов.

Таблица локальных дескрипторов содержит дескрипторы сегментов приложения, а таблица глобальных – дескрипторы сегментов программ операционной системы. Оставшиеся два бита селектора содержат код уровня привилегий сегмента, который проверяется при обращениях из других программ. Таким образом, реализуется защита сегментов.

14 бит селектора и 32 бита эффективного или исполнительного адреса, формируемого на основе машинной команды, объединяются в 46-ти разрядный *виртуальный адрес*.

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

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

Схема 64-х разрядной адресации выглядит аналогично, но для формирования 64-х адресов используются 64-х разрядные регистры.

Далее в основном будем рассматривать 32-х разрядные процессоры, однако практически все сказанное про них с учетом изменения размерности регистров может быть применено и к 64-х разрядным процессорам.

Модель памяти *Flat*. При создании приложений для защищенного режима в основном используется модель памяти *Flat* (пер. с англ. плоская). Полная схема адресации памяти в защищенном режиме используется только для небольшого подмножества управляющих программ операционной системы.

Модель *Flat* подразумевает, что каждому приложению из 4-х адресуемых Гб отводится линейное адресное пространство объемом 2 Гб, а остальные 2 Гб считаются предоставленными операционной системе. Базовый адрес в дескрипторах всех сегментов приложения устанавливается равным 0. В результате все сегменты приложения «перекрываются». Программа, данные и стек размещаются в разных местах памяти за счет различных смещений. Разделение памяти между приложениями осуществляется операционной

системой, которая размещает страницы приложений с одинаковыми линейными адресами в разных местах оперативной памяти (рисунок 10).

Рисунок 10 – Адресное пространство приложения

Следовательно, и защита сегментов при использовании модели *Flat* не работает. Однако использование указанной модели существенно упрощает процесс программирования.

1.3.3 Форматы машинных команд

Размер машинной команды процессора *IA*-32 колеблется от 1 до 15 байт. Структура команды представлена на рисунке 11.

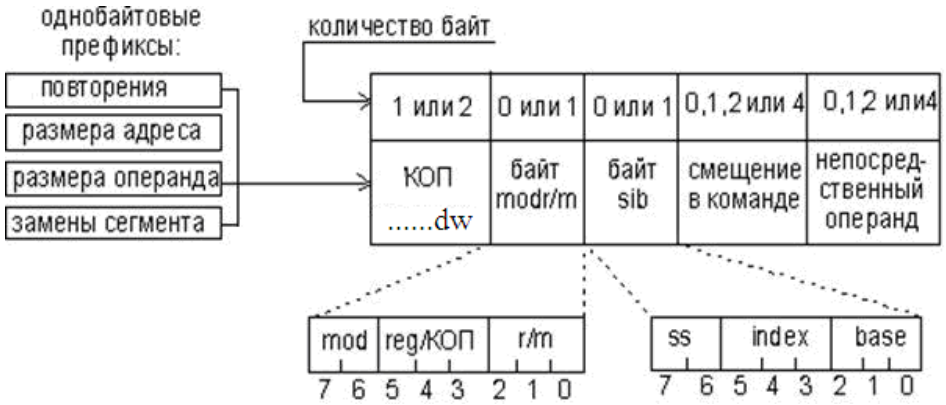


Рисунок 10 – Структура машинной команды *IA*-32

На рисунке использованы следующие обозначения:

d – бит направления обработки, например, пересылки данных: 1 – в регистр, 0 – из регистра; используется в арифметических командах и командах пересылки, если хотя бы один операнд находится в регистре;

w – размер операнда: 1 – операнды – двойные слова, 0 – операнды – байты;

mod – режим: 00 – *Disp*=0 – смещение в команде отсутствует (0 байт);

01 – *Disp*=1 – непосредственное смещение размером 1 байт;

10 – *Disp*=2 – непосредственное смещение размером 4 байта;

11 – оба операнда находятся в регистрах.

Регистры кодируются в зависимости от размера операнда (*w*):

	w=1		*w*=0	
reg	**000**	**EAX**	**000**	**AL**
(*r*)	**001**	**ECX**	**001**	**CL**
	010	**EDX**	**010**	**DL**
	011	**EBX**	**011**	**BL**
	100	**ESP**	**100**	**AH**
	101	**EBP**	**101**	**CH**
	110	**ESI**	**110**	**DH**
	111	**EDI**	**111**	**BH**

Как показано на рисунке помимо обязательного кода операции (КОП), иногда состоящего из двух частей, команда может включать от 0 до 4 однобайтовых префиксов, а также возможно байты адресации, непосредственного смещения (смещение, указанное в команде) и непосредственного операнда.

Префикс повторения – используется только для команд обработки строк и будет рассмотрен далее.

Префикс размера адреса (67*h*) – применяется для изменения размера смещения, например, если необходимо использовать смещение размером 16 бит при 32-х разрядной адресации.

Префикс размера операнда (66*h*) – указывается, если вместо 32-х разрядного регистра для хранения операнда используется 16-ти разрядный.

Префикс замены сегмента – используется при адресации данных любым сегментным регистром кроме **DS**.

Если в команде используется двухбайтовый регистр, например, **AX**, то перед командой добавляется префикс изменения длины операнда (66*h*).

Различают два вида команд, обрабатывающих операнд в памяти:

- команды без байта *sib* (англ. *scale-index-base* – масштаб–индекс–база);
- команды, содержащие байт *sib*.

Тип команды определяется по содержимому поля *m* в байт адресации (*r/m*): если $m \neq 100_2$, то байт *sib* в команде отсутствует и используется таблица 2. Иначе используется таблица 3, определяющая схемы адресации, которые формируются байтом *sib*.

Таблица 2 – Схемы адресации памяти в отсутствии байта *Sib*

Поле *r/m*	Эффективный адрес второго операнда		
	mod = 00B	*mod* = 01B	*mod* = 10B
000*B*	**EAX**	**EAX+Disp8**	**EAX+Disp32**
001*B*	**ECX**	**ECX+Disp8**	**ECX+Disp32**
010*B*	**EDX**	**EDX+Disp8**	**EDX+Disp32**
011*B*	**EBX**	**EBX+Disp8**	**EBX+Disp32**
100*B*	Определяется ***Sib***	Определяется ***Sib***	Определяется ***Sib***
101*B*	**Disp32**[1]	**SS:[EBP+Disp8]**	**SS:[EBP+Disp32]**
110*B*	**ESI**	**ESI+Disp8**	**ESI+Disp32**
111*B*	**EDI**	**EDI+Disp8**	**EDI+Disp32**

[1] – особый случай – адрес операнда не зависит от содержимого регистра EBP, а определяется только смещением в команде (прямая адресация).

Таблица 3 – Схемы адресации памяти при наличии байта *Sib*

Поле base	Эффективный адрес второго операнда		
	mod = 00B	mod = 01B	mod = 10B
000B	**EAX+ss*index**	**EAX+ss*index+Disp8**	**EAX+ss*index +Disp32**
001B	**ECX+ss*index**	**ECX+ss*index+Disp8**	**ECX+ss*index +Disp32**
010B	**EDX+ss*index**	**EDX+ss*index+Disp8**	**EDX+ss*index +Disp32**
011B	**EBX+ss*index**	**EBX+ss*index+Disp8**	**EBX+ss*index +Disp32**
100B	**SS:[ESP+ ss*index]**	**SS:[ESP+ss*index]+ Disp8**	**SS:[ESP+ss*index]+ Disp32**
101B	**Disp32**[1]**+ss*index**	**SS:[EBP+ss*index +Disp8]**	**SS:[EBP+ss*index +Disp32]**
110B	**ESI+ss*index**	**ESI+ss*index +Disp8**	**ESI+ss*index +Disp32**
111B	**EDI+ss*index**	**EDI+ss*index +Disp8**	**EDI+ss*index +Disp32**

[1] – особый случай – адрес операнда не зависит от содержимого регистра EBP, а определяется только смещением в команде (прямая адресация).

В таблице:
ss – масштаб; **index** – содержимое индексного регистра; **base** – содержимое базового регистра.

Примеры:

1) **`mov EBX,ECX`**

100010DW Mod Reg Reg
10001001 11 001 011
8 9 C B

Оглавление

2) **mov BX,CX**

префикс1 100010DW Mod Reg Reg
01100110 10001001 11 001 011
6 6 8 9 C B

3) **mov ECX,[DS:EBX+6]**

100010DW Mod Reg Reg См.мл.байт
10001011 01 001 011 00000110
8 B 4 B 0 6

4) **mov CX, [DS:EBX+6]**
префикс 100010DW Mod Reg Reg См.мл.байт
01100110 10001011 01 001 011 00000110
6 6 8 B 4 B 0 6

5) **mov CX, [ES:EBX+6]**
префикс1 префикс2 100010DW Mod Reg Reg См.мл.байт
01100110 00100110 10001011 01 001 011 00000110
6 6 2 6 8 B 4 B 0 6

6) **mov ECX, [EBX+EDI*4+6]**
100010DW Mod Reg Mem SS Ind Base См.мл.байт
10001011 01 001 100 10 111 011 00000110
8 B 4 C B B 0 6

При анализе кодов машинных команд следует иметь в виду, что команды, в качестве одного из операндов использующие регистры **AL**/**AX**/**EAX**, имеют специальный формат, который унаследован от еще более раннего предка – процессора *i*8080 (*Z*80). В этом процессоре регистр **AX** использовался как сумматор.

Указанные команды не содержат байта адресации и имеют коды операции $A0_{16}$ .. $A3_{16}$, два последних бита которых также расшифровываются, как *D* и *W*. При этом *D*=1 соответствует «из регистра», а *D*=0 – «в регистр», *W* имеет те же значения, что и в предыдущем случае. Сразу за кодом операции этих команд следуют 4-х байтовые (с учетом 32-х разрядной адресации) смещения.

Примеры:

1) **mov AL,[A]** **;** при адресе A соответствующем $403000

101000DW Смещение 32 разряда
10100000 00000000 00110000 01000000 00000000
A 0 0 0 3 0 4 0 0 0

2) **mov [B],AX** ; при адресе B соответствующем $403004

префикс1 101000DW Смещение 32 разряда

01100110 10100011 00000100 00110000 01000000 00000000

6 6 A 3 0 4 3 0 4 0 0 0

Команды процессоров IA-64 построены по аналогичному принципу.

Контрольные вопросы

1. Изобразите структурную схему процессора i8086.

Ответ.

2. Что собой представляет сегмент при 16-ти разрядной адресации?.

Ответ.

3. Назовите регистры общего назначения i8086? Как они были изменены в IA32?

Ответ.

4. Нарисуйте схему адресации защищенного режима.

Ответ.

5. Перечислите основные отличия машинных команд процессора *IA*-32 от машинных команд процессора *i*8086. Зачем они были выполнены?

Ответ.

2 ОСНОВЫ ПРОГРАММИРОВАНИЯ НА АССЕМБЛЕРЕ С ИСПОЛЬЗОВАНИЕМ ТРАНСЛЯТОРА *NASM*

2.1 Два вида синтаксиса языка ассемблера

Исторически сложилось так, что для языка ассемблера используют два вида синтаксиса *Intel* и *AT&T*. Различные способы записи предложений языка служили разработчикам для упрощения построения трансляторов. В таблице 4 показаны основные различия синтаксических конструкций, которые следует учитывать при написании и анализе исходных программ на ассемблере, а также при работе с отладчиком.

Таблица 4 – Основные различия двух видов синтаксиса языка ассемблера

Синтаксис *Intel*	**Синтаксис *AT&T***
1. Порядок записи источника и приемника <Приемник>, <Источник>	<Источник>, <Приемник>
`add eax, ebx`	`addl %ebx, %eax`
2. Мнемоника команды *AT%T* включает букву, обозначающую размер операндов (**`r`** – 8 байт, **`l`** – 4 байта, **`w`** – 2 байта, **`b`** – 1 байт):	
`add eax, ebx`	`addl %ebx, %eax`
3. Обозначение типа операнда в *AT&T* (**`%`** - содержимое регистра, **`$`** - непосредственное значение):	
`add eax, 5`	`addl $5, %eax`
4. Запись эффективного адреса [*BASE+INDEX*SCALE+DISP*]	*DISP*(*BASE,INDEX,SCALE*)
`mov eax,[ebx+ecx*4+a]`	`movl a(%ebx,%ecx,4),%eax`

Поскольку транслятор *Nasm* работает с исходными тестами программ, написанными с применением синтаксиса *Intel*, далее в настоящем пособии рассматривается именно этот вид синтаксиса.

Оглавление

2.2 Структура программы на языке ассемблера

Программа на языке ассемблера записывается по так называемому «свободному» формату, т.е. правила заполнения каких бы то ни было позиций строки специально не оговариваются.

В программе могут присутствовать предложения четырех типов:

- машинные команды ассемблера – такая команда преобразуется ассемблером в соответствующую машинную команду;
- директивы ассемблера – операторы управления процессами ассемблирования и компоновки;
- макрокоманды – заменяются на этапе предварительной обработки (макрогенерации) специально сгенерированной в соответствии с указанными параметрами совокупностью машинных команд;
- комментарии.

Машинные команды ассемблера имеют следующий формат:

[Метка :] Код операции [Список операндов] [; Комментарии].

В используемой для описания формата нотации квадратные скобки означают, что заключенная в них часть команды может отсутствовать. Код операции и список операндов разделяются хотя бы одним пробелом. Помимо двоеточия между меткой и командой, а также перед комментарием может быть произвольное количество пробелов. Операнды отделяются один от другого запятой. Точка с запятой в начале строки означает, что данная строка является строкой комментария. При необходимости можно использовать:

- символ переноса на следующую строку «\», например:

```
asdf    \
     DB    ' Пример использования символа переноса "\".'
```

- многострочный комментарий, который ограничивается символом, указанным после служебного слова comment, например:

```
COMMENT  $
        Это многострочный
        комментарий
$
```

NASM, как и другие ассемблеры, не различает строчные и прописные буквы ни в идентификаторах, ни в служебных словах.

Программа на ассемблере *NASM* состоит из сегментов (секций) следующих типов:

.text – сегмент кода, содержащий собственно текст программы;

.data – сегмент инициализированных данных, содержащий директивы объявление данных, для которых заданы начальные значения – память под эти данные распределяется во время ассемблирования программы;

.bss – сегмент неинициализированных данных, содержащий директивы объявление данных без задания начальных значений – память под эти данные отводится во время загрузки программы на выполнение.

Кроме этого программа включает сегмент стека, который формируется транслятором автоматически.

Для выполнения лабораторных работ в консольном режиме операционной системы *Astra Linux* специально созданы две заготовки программ на ассемблере.

Заготовка для 32-х разрядной программы выгладит следующим образом:

```
    section .data     ; сегмент инициализированных переменных
ExitMsg db "Press Enter to Exit",10 ; выводимое сообщение
lenExit equ $-ExitMsg
    section .bss      ; сегмент неинициализированных переменных
InBuf   resb    10    ; буфер для вводимой строки
lenIn   equ     $-InBuf
    section .text     ; сегмент кода
    global  _start
_start:
    ; write
    mov     eax, 4      ; системная функция 4 (write)
    mov     ebx, 1          ; дескриптор файла stdout=1
    mov     ecx, ExitMsg    ; адрес выводимой строки
    mov     edx, lenExit    ; длина выводимой строки
    int     80h             ; вызов системной функции
    ; read
    mov     eax, 3          ; системная функция 3 (read)
```

```
    mov     ebx, 0          ; дескриптор файла stdin=0
    mov     ecx, InBuf      ; адрес буфера ввода
    mov     edx, lenIn      ; размер буфера
    int     80h             ; вызов системной функции
    ; exit
    mov     eax, 1          ; системная функция 1 (exit)
    xor     ebx, ebx        ; код возврата 0
    int     80h             ; вызов системной функции
```

Заготовка включает коды обращения к системным функциям для выполнения операций ввода с клавиатуры, вывода на экран и завершения программы.

Заготовка 64-х разрядной программы содержит те же операции, но использует для их вызова другие номера системных функций и регистры. Кроме того, для вызова функций операционной системы вместо прерывания **int 80h** в этом случае обычно используют команду системного вызова **syscall**.

Заготовка 64-разрядной программы на ассемблере:

```
    section .data    ; сегмент инициализированных переменных
ExitMsg db "Press Enter to Exit",10 ; выводимое сообщение
lenExit equ $-ExitMsg
    section .bss     ; сегмент неинициализированных переменных
InBuf   resb    10  ; буфер для вводимой строки
lenIn   equ     $-InBuf
    section .text  ; сегмент кода
    global  _start
_start:
    ; write
    mov     rax, 1          ; системная функция 1 (write)
    mov     rdi, 1          ; дескриптор файла stdout=1
    mov     rsi, ExitMsg    ; адрес выводимой строки
    mov     rdx, lenExit    ; длина строки
    syscall                 ; вызов системной функции
    ; read
```

```
mov     rax, 0          ; системная функция 0 (read)
mov     rdi, 0          ; дескриптор файла stdin=0
mov     rsi, InBuf      ; адрес вводимой строки
mov     rdx, lenIn      ; длина строки
syscall                 ; вызов системной функции
; exit
mov     rax, 60         ; системная функция 60 (exit)
xor     rdi, rdi        ; return code 0
syscall                 ; вызов системной функции
```

Эти заготовки будут использоваться как исходные тексты в 4-х первых лабораторных работах и первом домашнем задании.

2.3 Директивы определения данных и резервирования памяти

Примечание. Далее рассмотрение ведется применительно к 32-х разрядным программам, однако следует поонимать, что написание 64-х разрядных программ для процессоров *Intel*-совместимых типов отличается практически только использованием 64-х разрядных регистров для формирования адресов и хранения данных.

Все данные, используемые в программах на ассемблере, обязательно должны быть объявлены, и часть из них инициализированы. При этом для данных должен задаваться объем выделяемой памяти в байтах.

Директивы объявления инициализированных данных имеют следующий формат:

[<Имя>][:] [`times` <Константа>]<Директива>[<Список_инициализаторов>]

где <Имя> – имя поля данных, которое может не присваиваться;

`times` <Константа> — псевдо-инструкция повторения полей данных, константа определяет количество повторений;

<Директива> — команда, объявляющая тип описываемых данных:

- **`db`** — определить байт,
- **`dw`** — определить слово (2 байта),

- **dd** — определить двойное слово (4 байта),
- **dq** — определить счетверенное слово (8 байт),
- **dt** — определить 10 байт;

<Список_инициализаторов> — последовательность инициализирующих констант, указанных через запятую.

В качестве инициализаторов при описании данных применяются:

- целые константы [<Знак>]<Целое> [<Основание системы счисления>],

  например:
  - -43236 – целые десятичные числа,
  - 0*x*22, 23*h*, $0*AD* – целые шестнадцатеричные числа (если шестнадцатеричная константа начинается с буквы, то перед ней указывается 0),
  - 0111010*b* – целое двоичное число;
- вещественные числа [<Знак>] <Целое>.[Целое][*e*[<Знак>] [<Целое>],

  например: -2., 34.*e*-28;
- символьные константы в апострофах или кавычках, например: '*A*' или "*A*";

строковые константы в апострофах или кавычках, например, '*ABCD*' или "*ABCD*".

Примеры:

```
a       db    23    ; записать в байт число 23 и присвоить этому байту имя «a»
        dw    1234h ; записать по адресу a+2 шестнадцатеричное число
```

Примечание – Одной из особенностей процессоров *Intel x*86 является то, что при записи чисел размером более 1 байта в память младший байт записывается в поле с меньшим адресом, затем следует байт перед младшим и так до старшего. Например, в предыдущем примере, если резервирование памяти выполнялась по адресу 100, то по адресу 100 будет записано 34*h*, а по адресу 101 – 12*h*.

```
val1    db    255  ; записать в байт число 255₁₀ = 11111111₂ и назвать vall
lue3    dw   -128  ; записать в слово число -128 и назвать lue3
alu  times 10 dw 0   ; записать 10 нулей по два байта  и назвать alu
        db    10h    ; записать в байт шестнадцатеричное число 10₁₆ = 16₁₀
v5      db    100101B ; записать в байт двоичное число  и назвать v5
        db    23,23h,0ch ; записать в байты каждое из указанных чисел
```

```
sdk     db    "Hello",0  ; записать в память строку, потом 0 и назвать sdk
        dw    -32767     ; записать в слово заданное число
        dd    12345678h  ; записать в двойное слово шестнадцатеричное число
```

Директивы резервирования памяти под данные имеют следующий формат:

[<Имя>][:]<Директива><Количество_полей>

где <Имя> – имя поля данных, которое может не присваиваться;

<Директива> – команда, объявляющая тип данных, под размещение которых резервируется память:

- **resb** — зарезервировать байт,
- **resw** — зарезервировать слово (2 байта),
- **resd** — зарезервировать двойное слово (4 байта),
- **resq** — зарезервировать счетверенное слово (8 байт),
- **rest**— зарезервировать 10 байт;

<Количество_полей> – целое число, которое показывает, сколько полей данного типа необходимо зарезервировать.

Примеры:

```
A     resb    5    ; зарезервировать 5 байтов
A     resd    10   ; зарезервировать 10 двойных слов
```

2.4 Операнды команд ассемблера

Операнды команд ассемблера могут записываться непосредственно в команду, находиться в регистрах или в основной памяти,

Данные, непосредственно заданные в команде, называются *литералами*. Так, в команде

```
mov    AH,3    ;  3 – литерал.
```

При записи литералов используют те же форматы, что и при записи инициализирующихзначений (см. раздел 2.3).

Если операнды команд ассемблера находятся в регистрах, то в команде на соответствующих местах указывают имена регистров. Например, в приведенном выше примере **AH** – имя однобайтового регистра-аккумулятора.

Адресация операндов, расположенных в основной памяти, может быть *прямой* и *косвенной*. При использовании прямой адресации в команде указывается символическое имя поля памяти, содержащего необходимые данные, например:

```
inc  [OPND]  ;
```
где OPND – символическое имя поля памяти, определенного директивой определения полей памяти ассемблера, например

```
OPND  DW  25
```

При трансляции программы ассемблер заменит символическое имя смещением поля данных относительно начала сегмента, т.е. определит непосредственное смещение и поместит его в команду, например

```
inc  word [28h] ;
```
увеличить на 1 слово со смещением 28h

Примечание – В этом случае адресация данных выполняется по схеме:

[<Содержимое регистра **EBP>** + <Смещение, заданное в команде>],

но на самом деле содержимое регистра **EBP** в вычислении исполнительного адреса не участвует, поскольку это частный случай команды, использующийся для явной адресации.

Размер операнда – слово, определяется директивой определения поля **DW.**

В отличие от прямого косвенный адрес определяет не смещение данных в основной памяти, а местоположение компонентов адреса этих данных. В этом случае в команде указываются один или два регистра в соответствии с допустимыми схемами адресации и непосредственное смещение, которое может задаваться числом или символическим именем.

Исполнительный адрес операнда при 32-х разрядной адресации считается по формуле:

EA = <База> + <Индекс>*<Масштаб> + <Непосредственное смещение>

	База		Индекс		Масштаб		Смещение
	EAX						
CS:	EBX		EAX				
SS:	ECX		EBX		1		отсутств.,
DS:	EDX	+	ECX	*	2	+	8 или
ES:	EBP		EDX		4		32бита
FS:	ESP		EBP		8		
GS:	ESI		ESI				
	EDI		EDI				

При отсутствии сегментного регистра в записи исполнительного адреса в команде по умолчанию считается, что номер дескриптора хранится в регистре **DS**.

Примеры:

```
mov     [ES:ECX],EDX            ; задана только база
mov     EAX,[ESI*4+TABLE]       ; заданы индекс и масштаб
```

При трансляции программы ассемблер определяет используемую схему адресации и соответствующим образом формирует машинную команду. При этом символическое имя заменяется непосредственным смещением относительно начала сегмента так же, как в случае прямой адресации.

Так, если в команде указано [**a**+**EBX**], то исполнительный адрес операнда определяется суммой содержимого регистра **EBX** и непосредственного смещения, заданного символическим именем «**a**», а, если указано [**EBP**+**ESI**+**6**], то исполнительный адрес – сумма содержимого регистров **EBP**, **ESI** и непосредственного смещения, равного 6.

Примечание. При использовании косвенной адресации по схеме

[<Содержимое регистра **EBP**> + <Смещение, заданное в команде>]

нулевое смещение не может быть опущено, так как частный случай адресации по соответствующей схеме с нулевой длиной смещения используется для организации прямой адресации (см. предыдущую страницу). Следовательно, при отсутствии смещения в команде следует указывать нулевое смещение, т.е. [**EBP** + **0**] .

Длина операнда может определяться:

а) кодом команды – в том случае, если используемая команда обрабатывает данные определенной длины, что специально оговаривается;

б) объемом регистров, используемых для хранения операндов (1, 2 или 4 байта);

в) специальными указателями **byte** (1 байт), **word** (2 байта) и **dword** (4 байта), **qword** (8 байтов) которые используются в тех случаях, если ***ни один операнд не находится в регистре и размер операнда отличен от размера, определенного директивой объявления данных***. Например:

```
        mov   byte [x], 255 ;  нас интересует только первый байт слова x
       . . .
x       DW    25
```

2.5 Команды пересылки / преобразования данных

При описании команд ассемблера использованы следующие условные обозначения:

***r*8** – один из 8-ми разрядных регистров: **AL**, **AH**, **BL**, **BH**, **CL**, **CH**, **DL**, **DH**;

***r*16** – один из 16-ти разрядных регистров: **AX**, **BX**, **CX**, **DX**, **SI**, **DI**, **SP**, **BP**;

***r*32** – один из 32-х разрядных регистров: **EAX**, **EBX**, **ECX**, **EDX**, **ESI**, **EDI**, **ESP**, **EBP**;

reg – произвольный регистр общего назначения любого размера;

sreg – один из 16-разрядных сегментных регистров: **CS**, **DS**, **ES**, **SS**, **FS**, **GS**;

***imm*8** – непосредственно заданное 8-ми разрядное значение;

***imm*16** – непосредственно заданное 16-ти разрядное значение;

***imm*32** – непосредственно заданное 32-х разрядное значение;

imm – непосредственно заданное значение любого размера;

***r*/*m*8** – 8-ми разрядный операнд в регистре или в памяти;

***r*/*m*16** – 16-ти разрядный операнд в регистре или в памяти;

***r*/*m*32** – 32-ти разрядный операнд в регистре или в памяти;

mem – адрес 8-ми, 16-ти или 32-х разрядного операнда в памяти;

***rel*8, *rel*16, *rel*32** – 8-ми, 16-ти или 32-х разрядная метка.

1. Команда пересылки данных – пересылает операнд размером 1, 2 или 4 байта из источника в приемник (рисунок 12):

MOV Приемник, Источник

Допустимые варианты:

mov *reg*, *reg*

mov *mem*, *reg*

mov *reg*, *mem*

mov *mem*, *imm*

mov *reg*, *imm*

mov *r*/*m*16, *sreg*

mov *sreg*, *r*/*m*16

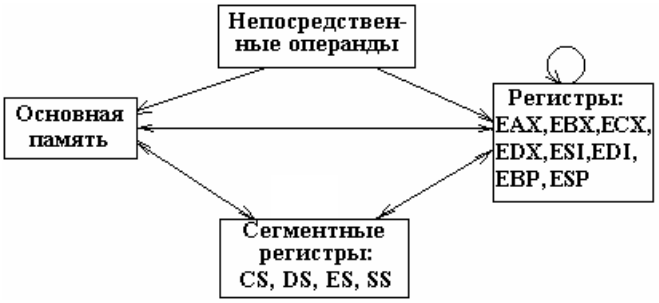


Рисунок 11 – Возможные пересылки командой **MOV**

Примеры:

```
mov     AX,BX       ; переписать число из AX в BX
mov     ESI,1000    ; записать число 1000 в ESI
mov     0[EDI],AL   ; переписать число из AL в память
mov     AX,code     ; записать число, определяемое сег. именем code, в AX
mov     DS,AX       ; переписать число из AX в DS
```

2. Команда перемещения и дополнения нулями **–** значение источника помещается в младшие разряды, а в старшие – заносятся нули**:**

```
MOVZX  Приемник, Источник
```

Допустимые варианты:

movzx *r*16/*r*32, *r*/*m*8

movzx *r*32, *r*/*m*16

Примеры:

а) **movzx EAX**, **BX** ; в AX заносится BX, в старшую часть EAX заносятся нули

б) **movzx SI**, **AH**

3. Команда перемещения и дополнения знаковым разрядом – команда выполняется аналогично предыдущей, но в старшие разряды заносятся знаковые биты:

```
MOVSX  Приемник, Источник
```

4. Команда обмена данных – команда меняет содержимое операндов местами.

```
XCHG  Операнд1, Операнд 2
```

Допустимые варианты:

xchg *reg, reg*

xchg *mem, reg*

xchg *reg, mem*

5-6. Команды записи слова или двойного слова в стек и извлечения из стека

PUSH *imm16* **/** *imm*32 **/** *r*16 **/** *r*32 **/** *m*16 **/** *m*32 ; запись

POP *r*16 **/** *r*32 **/** *m*16 **/** *m*32 ; извлечение

Команды автоматически изменяют содержимое **ESP**. Если в стек помещается 16-ти разрядное значение, то значение **ESP**:= **ESP**-2, если помещается 32 разрядное значение, то **ESP** := **ESP**-4.

Если из стека извлекается 16-ти разрядное значение, то значение **ESP**:= **ESP**+2, если помещается 32 разрядное значение, то **ESP**:= **ESP**+4.

Примеры:

```
push    SI
pop     word ptr [EBX]
```

8-9. Команды сложения – складывает операнды, а результат помещает на место первого операнда. В отличие от **ADD** команда **ADC** добавляет к результату значение бита флага переноса **CF**. Команда устанавливает флаги **CF**, **OF**, **ZF**, **SF** и др.

```
ADD  Операнд1, Операнд2
ADC  Операнд1, Операнд2
```

Допустимые варианты:

```
add   reg, reg
add   mem, reg
add   reg, mem
```

```
add   mem,imm
add   reg,imm
```

Пример:

add AX,BX ; складывает содержимое регистров AX и BX и помещает сумму в AX

10-11. Команды вычитания – вычитает из первого операнда второй и результат помещает по адресу первого операнда. В отличие от **SUB** команда **SBB** вычитает из результата значение бита флага переноса *CF*. Допустимые варианты те же, что и у сложения. Команда устанавливает флаги *CF*, *OF*, *ZF*, *SF* и др.

```
SUB  Операнд1, Операнд2
SBB  Операнд1, Операнд2
```

Пример:

sub AX,5 ; вычитает из содержимого AX число 5 и помещает результат в AX

13. Команда сравнения

```
CMP   Операнд1, Операнд2
```

Команда выполняется как команда вычитания, но, в отличие от нее, не запоминает результат, а только устанавливает флаги во флажковом регистре.

14-15. Команды добавления/вычитания единицы

```
INC   reg/mem
DEC   reg/mem
```

Примеры:

```
inc   AX
dec   byte [EBX+EDI+8]
```

16-17. Команды умножения

В команде указывается второй операнд. Первый операнд необходимо заранее занести в регистры **AL**/**AX**/**EAX** в зависимости от модификации команды: умножение байтов, слов или двойных слов. Результат имеет удвоенную длину и помещается в два регистра.

```
MUL    <Операнд2> ;  умножение беззнаковых (всегда положительных) чисел
IMUL   <Операнд2> ;  умножение чисел с учетом знака
```

Допустимые варианты:

mul/**imul** *r*|*m*8 ; AX= AL*<Операнд2>

mul/**imul** *r*|*m*16 ; DX:AX= AX*<Операнд2>

mul/**imul** *r*|*m*32 ; EDX:EAX= EAX*<Операнд2>

В качестве второго операнда нельзя указать непосредственное значение!!!

Пример:

```
mov   AX,4
imul  word [A]   ;  DX:AX:=AX*A
```

18-21. Команды «развертывания» чисел – операнды в команде не указываются. Операнд и его длина определяются кодом команды и не могут быть изменены. При выполнении команды происходит расширение записи числа до размера результата посредством размножения знакового разряда.

Команды часто используются при программировании деления чисел одинаковой размерности для получения делимого удвоенной длины.

```
CBW      ; байт в слово AL -> AX
CWD      ; слово в двойное слово AX -> DX:AX
CDQ      ; двойное слово в учетверенное EAX -> EDX:EAX
CWDE     ; слово в двойное слово AX -> EAX
```

Примеры:

```
cbw      ; чистит содержимое регистра AH знаковым разрядом регистра AL
```

22-23. Команды деления

Команда деления реализована аналогично команде умножения. Первый операнд должен иметь длину вдвое больше второго и должен быть заранее помещен в регистры **AX** / **DX:AX** / **EDX:EAX** в зависимости от того, какой вид деления выполняется: деление слова на байт, двойного слова на слово или учетверенного слова на двойное слово соответственно. Деление – целочисленное, поэтому получаем результат и остаток: результат в **AL**/**AX**/**EAX** и остаток – в **AH**/**DX**/**EDX**.

```
DIV    <Операнд2> ;  деление беззнаковых (всегда положительных) чисел
IDIV   <Операнд2> ;  деление чисел с учетом знака
```

Допустимые варианты:

div/idiv *r|m*8 ; AL= AX:<Операнд2>, AH – остаток

div/idiv *r|m*16 ; AX= (DX:AX):<Операнд2>, DX – остаток

div/idiv *r|m*32 ; EAX= (EDX:EAX):<Операнд2>, EDX – остаток

В качестве второго операнда нельзя указать непосредственное значение!!!

Пример:

```
mov   AX,40       ; загрузка делимого
cwd               ; развертывание делимого до 4-х байт в DX:AX
idiv  word [A]    ; деление  AX:=(DX:AX):A, в DX – остаток
```

Пример. Разработать программу, вычисляющую $X = (A+B)(B-1)/(D+8)$.

Ниже показан текст, который добавляется к шаблону.

В сегментах инициированных и неинициированных данных определяем все встречающиеся переменные:

```
          .data
A         SWORD 25
B         SWORD -6
D         SWORD 11
          .bss
X         RESW  1
```

Примечание. Использование сегмента неинициализированных данных **.bss** не является обязательным, все переменные инициализированные и неинициализированные можно объявить в сегменте инициализированных данных.

В сегменте кода записываем фрагмент вычисляющей программы:

```
          .text
Start:    mov     CX,[D]
          add     CX,8      ; CX:=D+8
          mov     BX,[B]
          dec     BX        ; BX:=B-1
          mov     AX,[A]
          add     AX,[D]     ; AX:=A+D
```

```
imul    BX        ; DX:AX:=(A+D)*(B-1)
idiv    CX        ; AX:=(DX:AX):CX
mov     [X],AX
. . .
```

Контрольные вопросы

1. Какие типы операторов могут использоваться в программах на ассемблере?

Ответ.

2. Какие типы сегментов включаются в программу на ассемблере? Что содержит каждый сегмент?

Ответ.

3. Данными каких типов может оперировать программа на ассемблере?

Ответ.

4. Какие типы операндов применяются в командах ассемблера? Как определяется длина этих данных?

Ответ.

5. Назовите команды ассемблера, которые выполняют операции сложения и вычитания чисел? Какие ограничения накладываются на размещение их операндов?

Ответ.

6. Назовите команды ассемблера, которые выполняют операции умножения и деления чисел? Какие ограничения накладываются на размещение и длину их операндов?

Ответ.

3 КОМАНДЫ ПЕРЕДАЧИ УПРАВЛЕНИЯ. ОСНОВНЫЕ ПРИЕМЫ ПРОГРАММИРОВАНИЯ

В языке ассемблера отсутствуют операторы, реализующие основные алгоритмические конструкции, такие как ветвление и циклы. Указанные конструкции моделируются с использованием машинных команд условной и безусловной передачи управления, а также команд сравнения, организации счетного цикла и некоторых других.

3.1 Команда безусловного перехода (аналог GOTO)

Команда безусловной передачи управления имеет следующий формат:

`JMP Адрес перехода`

Команда имеет несколько модификаций в зависимости от длины адресной части, так в модели *flat*:

`short` – используется при переходе по адресу, который находится на расстоянии -128..127 байт относительно адреса данной команды (длина адресной части команды перехода 1 байт);

`near` – при переходе по адресу, который находится в том же сегменте (длина адресной части 4 байта);

`far` – при переходе по адресу, который находится в другом сегменте (длина адресной части 6 байт).

При указании перехода к командам, предшествующим команде перехода, ассемблер сам определяет расстояние до метки перехода и строит адрес нужной длины. При программировании перехода к последующим частям программы необходимо для коротких переходов вставлять описатель **`short`** для экономии памяти. Указывать ближний переход не обязательно, поскольку в пределах модели памяти *flat* все адреса находятся в том же сегменте, т. е. предполагают вариант **`near`**, что и подразумевается по умолчанию.

В качестве адреса перехода помимо символических имен машинных команд ассемблера могут использоваться метки трех видов:

- <Имя > : **`nop`** ; **nop** – команда «нет операции»
- <Имя> **label near** ; метка для внутрисегментных переходов

- <Имя> **label far** ; метка для внесегментных переходов

Примеры:

jmp short b ; переход по адресу b

jmp [EBX] ; переход по адресу в регистре EBX (адрес определяется косвенно)

b label near ; описание метки перехода «b»

3.2 Команды условного перехода

Команды условного перехода используются после команд сравнения и арифметических команд. Для принятия решения о том, осуществлять или нет переход, команды перехода анализируют различные комбинации флагов флажкового регистра, установленные при выполнении предыдущих команд.

Формат любой команды условного перехода выглядит следующим образом:

Мнемоническая команда Адрес перехода

Мнемоника наиболее используемых команд условного перехода:

JZ – переход по «ноль» – *ZF*=1;

JE – переход по «равно» – *ZF*=1;

JNZ – переход по «не нуль» – *ZF*=0;

JNE – переход по «не равно» – *ZF*=0;

JL – переход по «меньше» – **SF**=1;

JNG, JLE – переход по «меньше или равно» – *SF*=1 или *ZF*=1;

JG – переход по «больше» – *SF*=0;

JNL, JGE – переход по «больше или равно» – *SF*=0 или *ZF*=1;

JA – переход по «выше» (беззнаковое «больше»);

JNA, JBE – переход по «не выше» (беззнаковое «не больше»);

JB – переход по «ниже» (беззнаковое «меньше»);

JNB, JAE – переход по «не ниже» (беззнаковое «не меньше»).

Формат команды рассчитан на короткие передачи управления (-128..127 байт) и передачи управления в пределах сегмента. Межсегментные условные переходы запрещены. Если смещение выходит за указанные пределы, то используется специальный прием:

вместо **jz zero** программируется:

```
                jnz   continue
                jmp   zero
continue:   ...
```

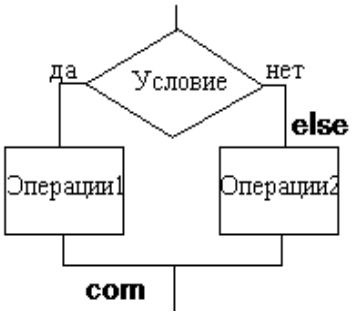
Если флаг нуля установлен (*ZF*=1), то мы пропускаем условный переход и выполняем безусловный, а если сброшен, то выполняем условный переход, обходя безусловный.

3.2.1 Программирование ветвлений

Ветвления программируются с использованием команд условной и безусловной передачи управления.

Сначала выполняем сравнение. В результате будут установлены флаги. Затем, если условие не выполняется, то переходим на метку **else**. Если условие выполняется, то переход не осуществляется, и управление переходит к следующей команде, т.е. выполнению команд, помеченных как Операции1. По завершению Операций1 передаем управление на команду, следующую за ветвлением, иначе будут выполняться команды, помеченные как Операции2, переход на которые был обозначен меткой **else**. Если переход был осуществлен, то после Операций 2 переходим на команду, следующую за ветвлением:

```
        cmp     ...
        j<условие>  else
        Операции1
        jmp     COM
ELSE:   Операции2
COM:        ...
```



Пример. Написать фрагмент вычисления ***X=max(A,B)***:

```
mov     ax,[A]
cmp     ax,[B]      ; сравнение A и B
jl      less        ; переход по меньше
mov     [X],ax
jmp     continue    ;  переход на конец ветвления
```

```
less:   mov    ax,[B]
        mov    [X],ax
continue: …
```

3.2.2 Программирование итерационных циклов (цикл-пока)

Программирование циклических процессов осуществляется с использованием либо команд переходов, либо – в случае счетных циклов – с использованием команд организации циклов.

Так, чтобы реализовать цикл-пока необходим один условный и один безусловный переходы:


```
cycl:   cmp   ...    ; проверка условия выхода
        jne   com    ; выход из цикла
        Операции     ; тело цикла
        jmp   cycl   ; возврат в цикл
com:     ...
```

Пример. Написать фрагмент суммирования чисел от 1 до 10, используя итерационный цикл.

```
        mov    ax,0     ;  обнуление суммы
        mov    bx,1     ;  первое слагаемое
cycl:   cmp    bx, 10   ;  слагаемое больше 10
        jg     continue ; выход из цикла
        add    ax, bx   ;  суммирование
        inc    bx       ;  следующее число
        jmp    cycl     ;  возврат в цикл
continue:  …            ; выход,  сумма - в ax
```

3.3 Команды организации циклической обработки. Организация счетных циклов

В качестве счетчика цикла во всех командах циклической обработки используется регистр **ECX**.

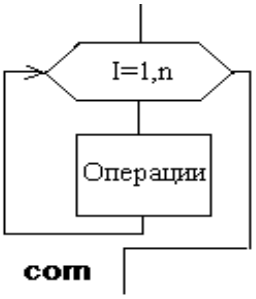
***1.* Команда организации счетного цикла:**

LOOP Адрес перехода ; переход возможен в пределах -128..127 байт

При каждом выполнении команда уменьшает содержимое регистра **ECX** на единицу и передает управление по указанному адресу, если **ECX** не равно 0.

Организация счетного цикла. Для организации счетного цикла с использованием команды **LOOP** необходимо записать количество повторений в регистр счетчика **ECX**. Тогда команда **LOOP** будет отсчитывать повторения, вычитая 1 из счетчика.

Примечание. Если перед началом цикла в регистр **ECX** загружен 0, то цикл выполняется 2^{32} раз. Такая ситуация называется «зацикливанием», поскольку программа надолго «зависает».




```
        mov   ECX,[n]        ; загрузка счетчика
begin_loop: Операции    ; тело цикла
        loop   begin_loop
```

Пример. Написать фрагмент суммирования чисел от 1 до 10, используя счетный цикл.

```
        mov     AX,0      ; обнуление суммы
        mov     BX,1      ; первое слагаемое
        mov     ECX,10    ; загрузка счетчика
cycl:   add     AX,BX     ; суммирование
        inc     BX        ; следующее число
        loop    cycl      ;  возврат в цикл
continue: …               ; выход,  сумма – в ax
```

Оглавление

***2.* Команда перехода по обнуленному счетчику.**

JCXZ Адрес перехода

Команда передает управление по указанному адресу, если содержимое регистра **ECX** равно 0.

Организация счетного цикла с проверкой счетчика.

```
        mov   ECX,[loop_count]  ;  загрузка счетчика
        jcxz  end_of_loop       ;  проверка счетчика
begin_loop: Операции  ; тело цикла
        loop   begin_loop
end_of_loop:   ...
```

3. Команды организации цикла с условием.

LOOPE Адрес перехода

LOOPNE Адрес перехода

При выполнении обеих команд содержимое регистра **ECX** уменьшается на единицу, после чего они передают управление по указанному адресу при условии, что содержимое **ECX** отлично от нуля, причем **LOOPE** дополнительно требует наличия флага «равно» (*ZF*=1), а **LOOPNE** – «не равно» (*ZF*=0).

Организация цикла со сложным условием. Конструкция Цикл со сложным условием позволяет эффективно реализовать поиск данных:

```
        mov   ECX,[loop_count]  ;  загрузка счетчика
        jcxz  end_of_loop       ;  проверка счетчика
begin_loop: Операции  ; тело цикла
        cmp   al,100   ; проверка содержимого al
        loopne  begin_loop
 end_of_loop:  ...
```

3.4 Команда загрузки исполнительного адреса

Команда загрузки исполнительного адреса имеет следующий формат:

LEA ***reg, mem***

В результате выполнения команды в регистр reg заносится исполнительный адрес операнда *mem*, размещенного в оперативной памяти.

Операнд *mem* обычно задается следующим образом:

Непосредственное смещение [База, Индекс*Масштаб] ,

причем любая часть описания может быть опущена, а непосредственное смещение может быть записано в скобках или в виде символического имени.

Возможны следующие варианты:

База		Индекс		Масштаб		Смещение
EAX						
EBX		**EAX**				
ECX		**EBX**		**1**		**отсутств.,**
EDX	**+**	**ECX**	*****	**2**	**+**	**8,16 или**
EBP		**EDX**		**4**		**32 бита**
ESP		**EBP**		**8**		
ESI		**ESI**				
EDI		**EDI**				

Исполнительный адрес рассчитывается по формуле:

EA = (База) + (Индекс)*Масштаб + Непосредственное смещение

где (…) - содержимое указанного регистра.

Примеры:

```
lea   EDX,[ECX]            ; в EDX загружается число по адресу в ECX
lea   EBX,[ESI*4+TABLE]    ; в EBX загружается  число из ESI, умноженное на 4,
                           ; плюс смещение символического имени Table
lea   EBX,[Exword]         ; в EBX загружается смещение символического имени
                           ; Exword относительно начала сегмента
lea   EBX,[EDI+10]         ; в  EBX загружается адрес l0-го байта относительно
                           ; точки, на которую указывает адрес в регистре EDI.
```

Команда **lea** обычно используется для задания адресов массивов, матриц и строк.

3.4.1 Обработка одномерных массивов

Массив во внутреннем представлении – это последовательность элементов в памяти. В ассемблере такую последовательность можно определить, например, так:

```
A       dw  10,13,28,67,0,-1   ;  массив из 6 чисел длиной слово
```

Программирование обработки выполняется с использованием адресного регистра, в котором хранится либо смещение текущего элемента относительно начала сегмента данных, либо его смещение относительно начала массива. При переходе к следующему элементу и то и то смещение увеличиваются на длину элемента. Если длина элемента отлична от единицы, то можно использовать масштаб.

Пример. Написать процедуру, выполняющую суммирование массива из 10 чисел размером слово.

Вариант 1 (используется адрес):

```
        mov   AX,0
        lea   EBX,[MAS]
        mov   ECX,10
CYCL:   add   AX,[EBX]
        add   EBX,2
        loop  CYCL
```

Вариант 2 (используется смещение):

```
        mov   AX,0
        mov   EBX,0
        mov   ECX,10
CYCL:   add   AX,[EBX*2+MAS]
        add   EBX,1
        loop  CYCL
```

Второй вариант позволяет получать более наглядный код и потому является предпочтительным.

В том случае, если элементы просматриваются не подряд, адрес элемента может рассчитываться по его номеру (числа нумерованы с единицы):

$A_{исп} = A_{начала}$+(<Номер> - 1)*<Длина элемента>.

Полученный по формуле адрес записывается в 32-х разрядный регистр и используется для доступа к элементу.

Пример. Написать фрагмент, который извлекает из массива, включающего 10 чисел размером слово, число с номером *n* ($n \le 10$).

```
mov   EBX,[N]            ; номер числа
dec   EBX                ; вычитаем 1
mov   AX, [EBX*2+MAS]    ; результат в AX
```

3.4.2 Обработка матриц

Значения матрицы могут располагаться в памяти по строкам и по столбцам. Для определенности будем считать, что матрица расположена в памяти построчно, как в языках Паскаль и C++.

При обработке элементов матрицы следует различать просмотр по строкам, просмотр по столбцам, просмотр по диагоналям и произвольный доступ.

Если матрица расположена в памяти по строкам и просмотр выполняется по строкам, то обработка может выполняться так, как в одномерном массиве, без учета перехода от одной строки к другой.

Пример. Написать фрагмент определения максимального элемента матрицы A(3,5).

```
        mov   EBX,0              ; номер элемента 0
        mov   ECX,14             ; счетчик цикла
        mov   AX,[A]             ; заносим первое число
CYCL:   cmp   AX,[EBX*2+2+A]     ; сравниваем числа
        jge   NEXT               ; если больше, то перейти к следующему
        mov   AX,[EBX*2+2+A]     ; если меньше, то запомнить
NEXT:   add   EBX,1              ; переходим к следующему числу
        loop  CYCL
```

Просмотр по строкам при необходимости фиксировать завершение строки и просмотр по столбцам при построчном расположении в памяти выполняются в двойном цикле.

Пример. Определить сумму максимальных элементов столбцов матрицы A(3,5).

```
        mov   AX,0               ; обнуляем сумму
        mov   EBX,0              ; смещение элемента столбца в строке
        mov   ECX,5              ;  количество столбцов
CYCL1:  push  ECX                ;  сохраняем счетчик
        mov   ECX,2              ;  счетчик элементов в столбце - 1
        mov   DX,[EBX+A]         ; заносим первый элемент столбца
        mov   ESI,10             ; смещение второго элемента столбца
CYCL2:  cmp   DX,[EBX+ESI+A]     ; сравниваем
        jge   NEXT               ; если больше или равно - к следующему
        mov   DX,[EBX+ESI+A]     ;  если меньше, то сохранили
NEXT:   add   ESI,10             ;  переходим к следующему элементу
        loop  CYCL2              ; цикл по элементам столбца
        add   AX,DX              ;  просуммировали максимальный элемент
```

```
pop    ECX                ; восстановили счетчик
add    EBX,2              ; перешли к следующему столбцу
loop   CYCL1              ; цикл по столбцам
```

При просмотре по диагонали обычно используют один цикл, через переменную которого рассчитываются смещения элементов массива. Однако проще использовать специальный регистр смещения, который должен соответствующим образом переадресовываться.

3.5 Команды обработки строк

Команды обработки строк используются для организации циклической обработки последовательностей элементов длиной 1, 2 или 4 байта. Адресация операндов при этом выполняется с помощью пар регистров: **DS:ESI** – источник, **ES:EDI** – приемник. Команды имеют встроенную корректировку адреса операндов согласно флагу направления *DF*: *DF*=1 – автоматическое уменьшение адреса на длину элемента, *DF*=0 – автоматическое увеличение адреса на длину элемента. Автоматическая корректировка осуществляется после выполнения операции.

Установка требуемого значения флага направления производится специальными командами: **STD** – установка флага направления в единицу,

CLD – сброс флага направления в ноль.

1. Команда загрузки строки LODS

LODSB (загрузка байта),

LODSW (загрузка слова),

LODSD (загрузка двойного слова),

Команда использует адрес операнда по умолчанию в **DS:ESI**. Она загружает байт в **AL**, слово в **AX** или двойное слово в **EAX.**

2. Команда записи строки STOS

STOSB (запись байта),

STOSW (запись слова),

STOSD (запись двойного слова)

Команда записывает в основную память содержимое **AL**, **AX** или **EAX** соответственно. Для адресации операнда используются регистры **ES:EDI**.

3. Команда пересылки MOVS

MOVSB (пересылка байта),

MOVSW (пересылки слова),

MOVSD (пересылки двойного слова).

Команда пересылает элемент строки из области, адресуемой регистрами **DS:ESI**, в область, адресуемую регистрами **ES:EDI**.

4. Команда сканирования строки SCAS

SCASB (поиск байта),

SCASW (поиск слова).

SCASD (поиск двойного слова).

По команде содержимое регистра **AL**, **AX** или **EAX** сравниваются с элементом строки, адресуемым регистрами **DS:DI,** и устанавливается значение флажков в соответствии с результатом **[DI] - AL** или **[DI]-AX**.

5. Команда сравнения строк CMPS

CMPSB (сравнение байт),

CMPSW (сравнение слов),

CMPSD (сравнение двойных слов).

По команде элементы строк, адресуемых парами регистров **DS:ESI** и **ES:EDI**, сравниваются и устанавливаются значения флажков в соответствии с результатом **[EDI]-[ESI]**.

6. Префиксная команда повторения.

```
REP  Команда
```

Команда позволяет организовать повторение указанной команды **ECX** раз.

Пример:

```
rep  stosb
```

Здесь поле, адресуемое парой регистров **ES:EDI** длиной **ECX** заполняется содержимым **AL**.

7. Префиксные команды «повторять, пока равно» и «повторять, пока не равно».

```
REPE  Команда
REPNE Команда
```

Префиксные команды используются совместно с командами **CMPS** и **SCAS**. Префикс **REPE** означает повторять, пока содержимое регистра **ECX** не равно нулю и значение флажка нуля равно единице, а **REPNE** – повторять, пока содержимое регистра **ECX** не равно нулю и значение флажка нуля равно нулю.

Контрольные вопросы

1. Напишите фрагмент программы, реализующей ветвление. Почему при написании фрагмента использованы команды условной и безусловной передачи управления?

Ответ.

2. Напишите фрагмент программы, реализующей итерационный цикл. Почему при написании фрагмента использованы команды условной и безусловной передачи управления?

Ответ.

3. Как в ассемблере моделируется обработка массивов и матриц? Почему?

Ответ.

4. В чем состоит особенность определения местонахождения операндов строковой обработки? С чем связана такая реализация?

Ответ.

СПИСОК ИСТОЧНИКОВ

1. Ирвин К. Язык ассемблера для процессоров Intel. – М.: Изд. дом «Вильямс», 2005.
2. Костюк Д.А., Четверкина Г.А. Программирование на ассемблере в GNU/Linux: методическое пособие. Брест: изд-во БрГТУ, 2013. 68 с. Способ доступа: https://www.bstu.by/uploads/attachments/metodichki/kafedri/EVMiS_Assembler_v_GNU-Linux.pdf.
3. Столяров А.В. Программирование на языке ассемблера NASM для ОС Unix: Уч. Пособие. - М.: МАКС Пресс, 2011. - 188 с. Способ доступа: http://www.stolyarov.info/books/pdf/nasm_unix.pdf.