Оглавление

МГТУ им. Н.Э. Баумана

Факультет «Информатика и Системы Управления»

Кафедра ИУ-6 «Компьютерные системы и сети»

ИВАНОВА ГАЛИНА СЕРГЕЕВНА,

НИЧУШКИНА ТАТЬЯНА НИКОЛАЕВНА

Основы программирования на ассемблере IA-32

Учебное пособие

МОСКВА

2010 год МГТУ им. Баумана

Оглавление

Аннотация

Настоящее учебное пособие ориентировано на студентов, начинающих изучать основы программирования на 32-х разрядном ассемблере с использованием транслятора MASM32. Оно содержит необходимые сведения об архитектуре процессоров фирмы Intel, информацию о структуре программы на ассемблере и основных директивах, а также сведения о форматах машинных команд и правилах их записи в ассемблере. Кроме этого в пособии обсуждаются некоторые приемы программирования на ассемблере, такие как программирование ветвлений, организация циклов разного вида и принципы программирования обработки массивов и матриц. Приведены также форматы и примеры использования некоторых команд API, предназначенных для организации ввода-вывода в консольном режиме Windows.

Пособие предназначено для студентов 2 курса кафедры Компьютерные системы и сети, изучающих дисциплину Системное программное обеспечение.

Оглавление

1 ВЫЧИСЛИТЕЛЬНЫЕ СИСТЕМЫ НА БАЗЕ ПРОЦЕССОРов i8086 – IA-32

1.1 Архитектура «с общей шиной»

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

ВС с процессорами Intel строится на базе архитектуры с общей шиной (см. рисунок 1). При этой архитектуре процессор соединяется со всеми остальными устройствами через общие шины управления, адреса и данных.

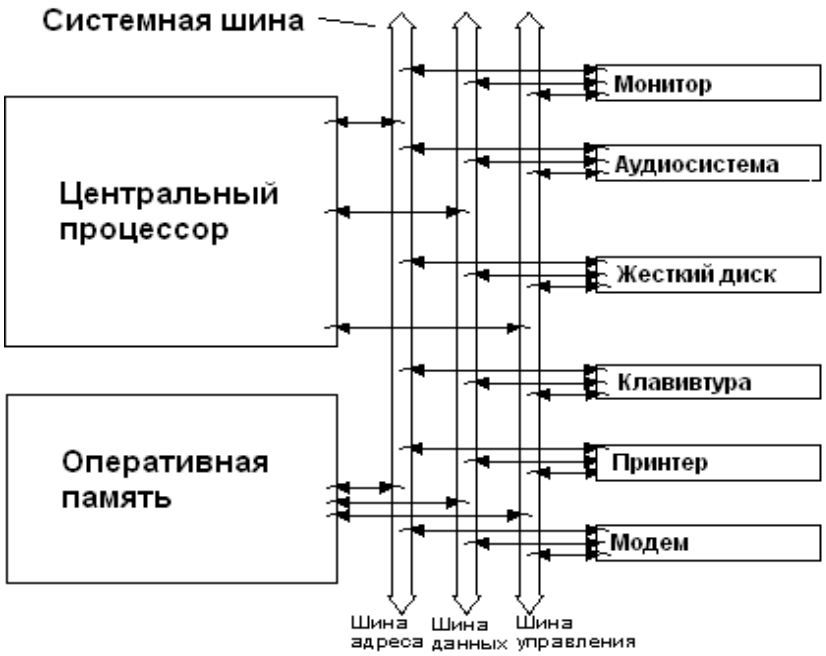


Рисунок 1 – Архитектура с общей шиной

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 обработку. В настоящее время этот недостаток преодолевается увеличением рабочей частоты системной шины до 1.6 ГГц.

Оглавление

1.2 Процессор i8086

1.2.1 Структурная схема процессора

Родоначальником современного семейства процессоров фирмы Intel является процессор i8086. На рисунке 2 представлена структурная схема этого процессора. В его состав входят: устройство управления (УУ), арифметико-логическое устройство (АЛУ), блок преобразования (формирования) адресов и регистры.

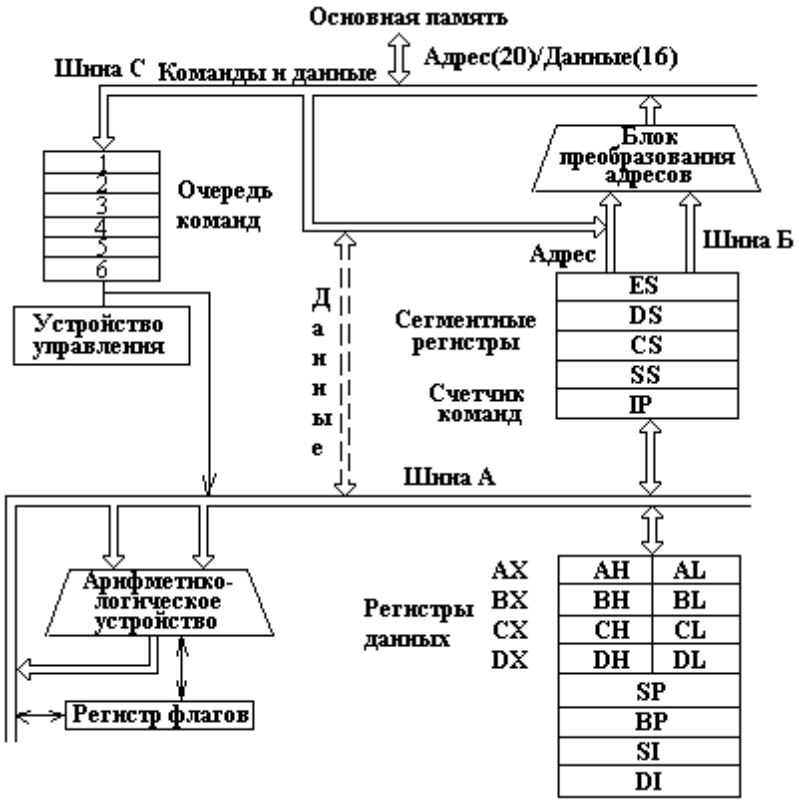


Рисунок 2 – Структура процессора i8086

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 используются для хранения управляющей информации: адресов и данных.

Оглавление

Всего в состав микропроцессора i8086 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 результата,

BP – регистр-указатель базы;

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.

Оглавление

1.2.2 Организация основной памяти ВС на базе процессоров i8086

Минимальной адресуемой единицей основной памяти является *байт*, состоящий из 8 бит. Адресовать отдельно бит нельзя. Если необходимо получить доступ к определенному биту, то сначала ищется соответствующий байт, а затем уже в нем – нужный бит.

Номер байта является его физическим адресом в устройстве памяти.

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 – сегмента кодов, т.е. собственно программы;
- DS, ES – сегмента данных;
- SS – 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 физический адрес данных (см. рисунок 3).

Рисунок 3 – Формирование 20-ти разрядного физического адреса

Таким образом, для адресации основной памяти в микропроцессоре i8086 предусматриваются 20-битовые адреса, что позволяет работать с основной памятью до 1 Мб.

Оглавление

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

Смещение для каждого типа сегмента формируется по своим правилам (см. 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.

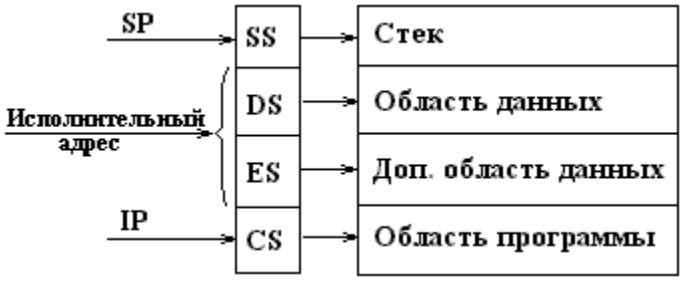


Рисунок 4 – Адресация различных сегментов сегментными регистрами

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 (см. рисунок 5). Таким образом, стек растет в область младших адресов, и в начальный момент времени указатель SP должен содержать максимально возможное для конкретного размера стека смещение.

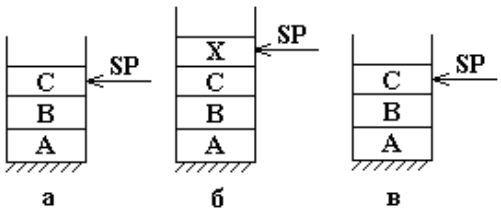


Рисунок 5 – Процессы записи в стек и чтения из стека:

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

Оглавление

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 (см. далее).

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 представлен флажковый регистр Flags процессора i8086, в котором в виде однобитовых признаков по принципу ДА – НЕТ (ВКЛЮЧЕНО – ВЫКЛЮЧЕНО) фиксируется информация о результатах выполнения машинных команд, например арифметических.

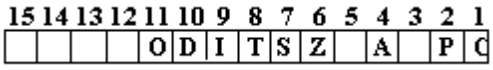


Рисунок 6 – Флажковый регистр

Основные флаги флажкового регистра имеют следующее значение:

OF – переполнение разрядной сетки;

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

Оглавление

IF – разрешение прерывания;

SF– признак знака: 1 – результат < 0, 0 – результат > 0

ZF – признак нуля: 1 – результат = 0

AF – признак наличия переноса из тетрады;

CF – признак переноса.

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

Оглавление

1.3 Программная модель процессора IA-32

Структура процессора семейства IA-32 очень сложна, поскольку в них аппаратно реализована совокупность параллельных конвейеров (конвейерная и суперскалярная архитектура). На рисунке 7 процессор IA-32 представлен в виде набора 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 последующих 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).

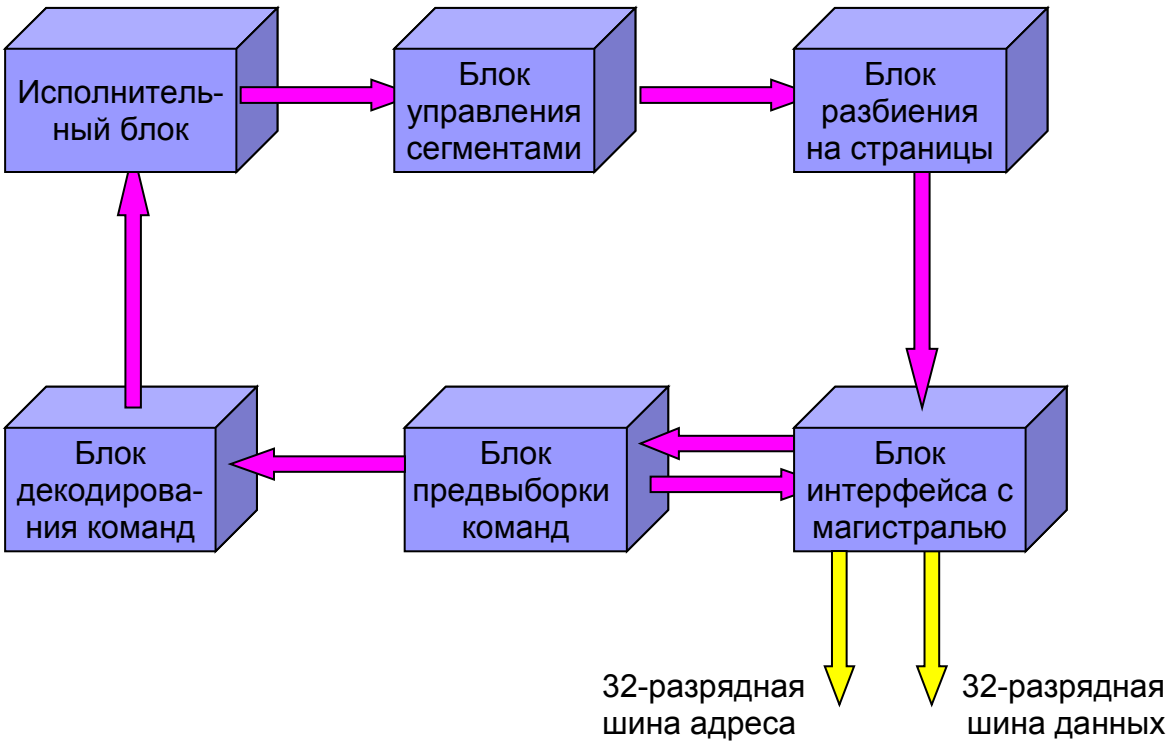


Рисунок 7 – Структура процессора семейства IA-32

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

Оглавление

1.3.1 Регистры общего назначения

Большинство регистров процессоров семейства IA-32 – 32-х разрядные. Они включают 16-ти разрядные регистры адресов и данных, имевшиеся в прародителе i8086, как младшую часть (см. рисунок 8), и обеспечивают доступ к ним по указанным именам.

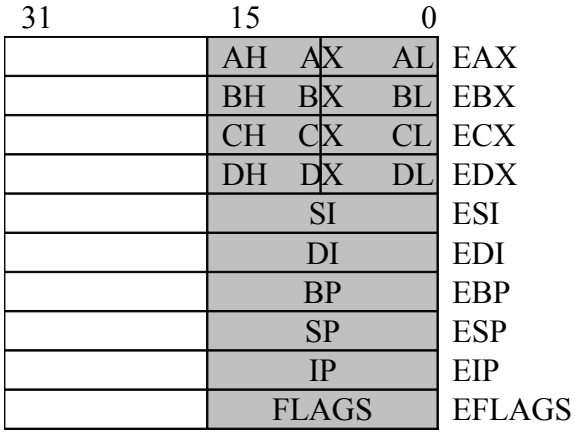


Рисунок 8 – Регистры адресов и данных IA-32

Сегментные регистры 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 (см. далее).

Кроме того было добавлено еще несколько специализированных регистров, используемых в защищенном режиме:

а) управляющие регистры C0..C3;

б) регистры системных адресов:

GDTR – регистр адреса таблицы глобальных дескрипторов;

LDTR – регистр адреса таблицы локальных дескрипторов;

IDTR – регистр адреса таблицы дескрипторов прерываний;

TR – регистр состояния задачи;

в) отладочные регистры;

г) тестовые регистры.

Оглавление

1.3.2 Режимы адресации. Схема адресации защищенного режима

Процессоры IA-32 поддерживают три режима адресации:

1.1 *реальный* – в этом режиме адрес формируется аналогично i8086, т.е. при формировании адреса используются 16-ти разрядные смещения и 16-ти разрядные сегментные адреса, которые хранятся в сегментных регистрах. При их сложении по приведенной выше схеме получаются 20-ти разрядные физические адреса, поэтому в этом режиме доступен только первый мегабайт оперативной памяти;

1.2 *защищенный* – в этом режиме используется 32-х разрядная адресация, предусматривающая несколько вариантов защиты, откуда и появилось название этого режима;

1.3 *виртуальный* – в этом режиме процессор моделирует псевдоодновременную работу нескольких виртуальных процессоров i8086. В настоящее время режим устарел и практически не используется.

Требование сохранить возможность выполнения программ, использующих 16-ти разрядную адресацию, привело к тому, что схема 32-х разрядной адресации является многокомпонентной (см. рисунок 9).

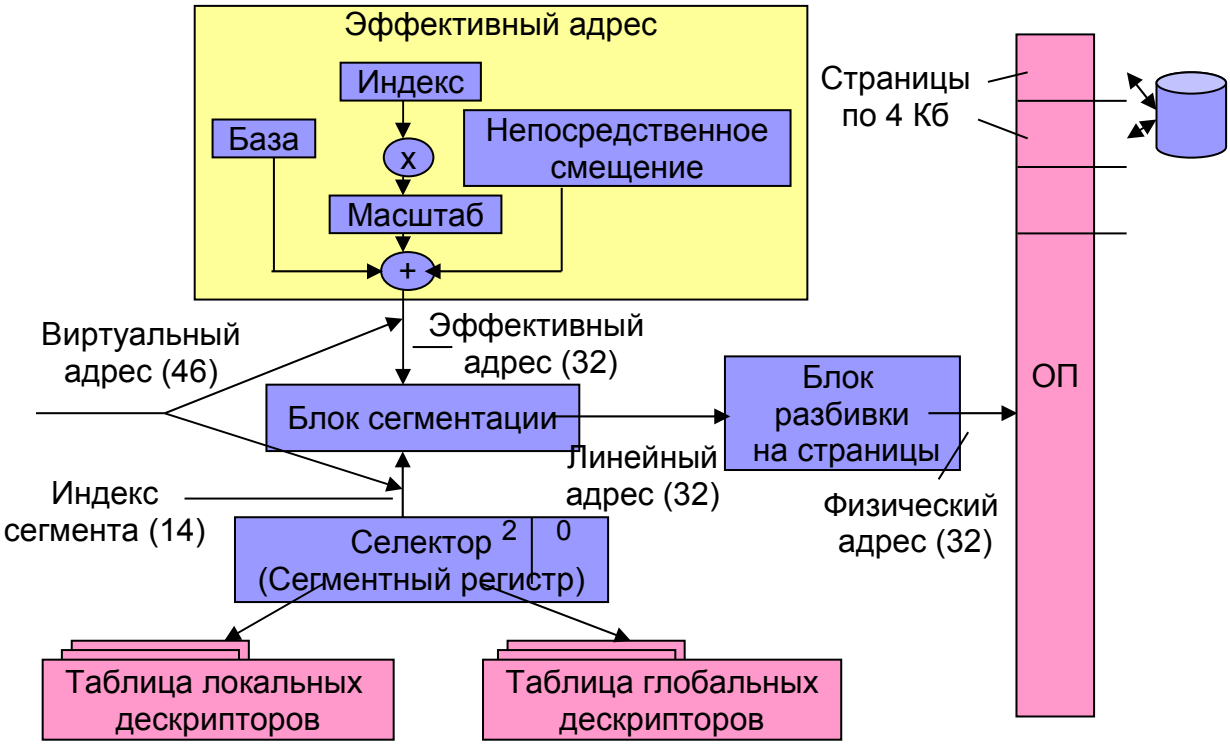


Рисунок 9 – Схема 32-х разрядной адресации в защищенном режиме

Оглавление

В этом режиме по-прежнему используется сегментная организация памяти, но размер сегмента уже не ограничивается 64 Кб, а теоретически может достигать 4 Гб. 32-х разрядный адрес базы сегмента хранится не в виде сегментного 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 – 64 Тб; линейное – 4 Гб; физическое – 4 Гб.

При создании приложений Windows в основном используется модель памяти **Flat** «плоская». Эта модель подразумевает, что каждому приложению отводится линейное адресное пространство объемом 2 Гб, а остальные 2 Гб предоставляются 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. Следовательно и защита сегментов при этой модели не работает.

Оглавление

1.3.3 Форматы машинных команд

Размер машинной команды процессора IA-32 колеблется от 1 до 15 байт. Структура команды представлена на рисунке 10. 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 (67h) – применяется для изменения размера смещения, например, если необходимо использовать смещение размером 16 бит при 32-х разрядной адресации.

Префикс размера операнда (66h) – 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.

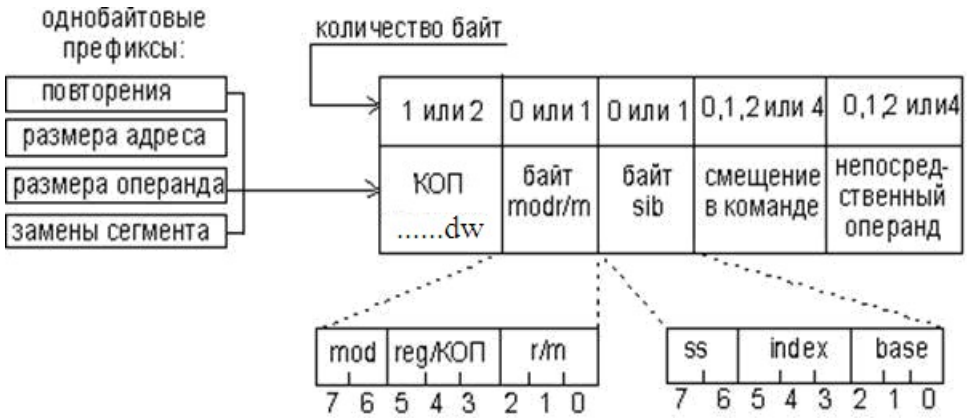


Рисунок 10 – Структура машинной команды IA-32

На рисунке 10 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;

Оглавление

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 2 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**

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

Различают два вида команд, обрабатывающих операнд в памяти:

- команды без байта sib (масштаб–индекс–база);
- команды, содержащие байт sib.

Тип команды определяется по содержимому поля m байта адресации (r/m): если m≠100, то байт sib в команде отсутствует и используется таблица 1. В противном случае используется таблица 2, определяющая схемы адресации, которые формируются байтом sib.

Таблица 1 – Схемы адресации памяти в отсутствии байта Sib

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

Оглавление

Таблица 2 – 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

В таблице:
ss – масштаб; **index** – содержимое индексного регистра; **base** – содержимое базового регистра;
[1] – особый случай – адрес операнда не зависит от содержимого регистра EBP, а определяется только смещением в команде (прямая адресация).

Кроме того, 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 i8080 (Z80). В этом процессоре регистр AX использовался как сумматор.

Примеры:
1) `mov EBX,ECX`

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

2) `mov BX,CX`

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

3) `mov ECX,DS:6[EBX]`

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

4) `mov CX,DS:6[EBX]`

префикс 100010DW Mod Reg Reg См.мл.байт
01100110 10001011 01 001 011 00000110
6 6 8 B 4 B 0 6

Оглавление

5) `mov CX,ES:6[EBX]`

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,6[EBX+EDI*4]`

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

Оглавление

Контрольные вопросы

1. Изобразите структурную схему процессора i8086.

Ответ.

2. Что собой представляет сегмент при 16-ти разрадной 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 IA32 от машинных команд процессора i8086. Зачем они были выполнены?

Ответ.

Оглавление

2 ОСНОВЫ ПРОГРАММИРОВАНИЯ НА АССЕМБЛЕРЕ С ИСПОЛЬЗОВАНИЕМ ТРАНСЛЯТОРА MASM32

2.1 Структура программы на языке ассемблера

Запись программы на языке ассемблера MASM32 выполняется по «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 машинную;
- 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 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 "\".'
```

Оглавление

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

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

Masm32, как и другие ассемблеры, не различает строчные и прописные буквы ни в идентификаторах, ни в служебных словах. Однако при работе в ассембере обычно устанавливают опцию различия строчных и прописных символов в идентификаторах (OPTION CASEMAP:NONE), поскольку эти различия существенны при вызове функций API.

Программа на ассемблере MASM32 состоит из сегментов следующих типов:

- **сегмент кода**, содержащий собственно текст программы;
- **сегменты данных**:
  - ***сегмент констант***, содержащий директивы объявления данных, изменение которых в программе не предполагается;
  - ***сегмент инициализированных данных***, содержащий директивы объявление данных, для которых заданы начальные значения – память под эти данные распределяется во время ассемблирования программы;
  - ***сегмент неинициализированных данных***, содержащий директивы объявление данных – память под эти данные отводится во время загрузки программы на выполнение;
- **сегмент стека**, определяемый для ассемблера по заданному размеру.

В программе сегменты описываются полными или сокращенными директивами.

Сокращенные директивы описания сегмента кодируются следующим образом:

1. **.CODE [Имя сегмента]** – начало или продолжение сегмента кода;
2. **.MODEL Модель [Модификатор][,Язык][,Модификатор языка]**

где **Модель** – определяет набор и типы сегментов; при 32-х разрядной адресации используется единственная модель FLAT;

Модификатор – определяет тип адресации: use16, use32, dos;

Язык и Модификатор языка – определяют особенности передачи параметров при вызове подпрограмм на разных языках C, PASCAL, STDCALL;

Оглавление

3. **.DATA** – начало или продолжение сегмента инициализированных данных;
4. **.DATA?** – начало или продолжение сегмента неинициализированных данных;
5. **.CONST** – начало или продолжение сегмента неизменяемых данных;
6. **.STACK [Размер]** – начало или продолжение сегмента стека.

В среде RadAsm специально для выполнения лабораторных работ создана заготовка консольной программы conapp.tpl, которая выглядит следующим образом:

```
        .586                    ; использование набора команд i80586
        .MODEL  flat, stdcall   ; модель памяти и тип передачи параметров
       OPTION CASEMAP:NONE      ; чувствительность идентификаторов к регистру
Include kernel32.inc            ; подключение файлов описаний библиотечных п/п
Include masm32.inc
IncludeLib kernel32.lib         ; подключение библиотек при компоновке
IncludeLib masm32.lib
          .DATA                 ; сегмент инициализированных данных
Msg        DB   "Press Enter to Exit",0AH,0DH,0
          .DATA?                ; сегмент неинициализированных данных
inbuf      DB   100 DUP (?)
          .STACK  4096          ; сегмент стека – 4096 байт
          .CODE                 ; сегмент кода
Start:
           XOR    EAX,EAX       ; очистка регистра
           Invoke StdOut,ADDR Msg  ; вызов процедуры вывода
           Invoke StdIn,ADDR inbuf,LengthOf inbuf
                                   ; вызов процедуры ввода
           Invoke ExitProcess,0  ; вызов процедуры завершения
           END    Start
```

Эта заготовка включает вызовы процедур ввода и вывода (см. комментарии), совокупность которых обеспечивает задержку закрытия окна консольного приложения до нажатия клавиши Enter.

Оглавление

2.2 Директивы определения полей памяти для размещения данных

Все данные, используемые в программах на ассемблере, обязательно должны быть объявлены. Директива объявления данных имеет следующий формат:

[Имя] Директива [Константа DUP (] Список инициализаторов [)]

где **Имя** – символическое имя поля данных, которое может не присваиваться;

Директива – команда, объявляющая тип описываемых данных (см. таблицу 3);

Таблица 3 – Директивы определения данных

Директива	Описание типа данных
BYTE	8-разрядное целое без знака
SBYTE	8-разрядное целое со знаком
WORD	16-разрядное целое без знака или ближний указатель реального режима
SWORD	16-разрядное целое со знаком
DWORD	32-разрядное целое без знака, дальний указатель реального режима или ближний указатель защищенного режима
SDWORD	32-разрядное целое со знаком
FWORD	48-разрядное целое или дальний указатель защищенного режима
QWORD	64-разрядное целое
TBYTE	80-разрядное целое
REAL4	32-х разрядное короткое вещественное
REAL8	64-х разрядное длинное вещественное
REAL10	80-ти разрядное расширенное вещественное

Примечание – В качестве директив также могут использоваться:

- **DB –** определить байт,
- **DW –** определить слово,
- **DD –** определить двойное слово (4 байта),
- **DQ –** определить четыре слова (8 байт),
- **DT –** определить пять слов (10 байт),

однако при их применении знаковые и беззнаковые, целые и вещественные типы не различаются, поэтому директивы считаются устаревшими, хотя реальный контроль типов данных в ассемблере в настоящее время не реализован.

Оглавление

Константа DUP – используется при описании повторяющихся данных, тогда константа определяет количество повторений;

Список инициализаторов – последовательность инициализирующих констант, указанных через запятую, или символ «?», если инициализирующее значение не определяется.

В качестве инициализирующих констант при описании данных применяются:

- целые константы формата

  [Знак]Целое[Основание системы счисления],

  например:

  - -43236, 236**d** – целые десятичные числа (применяется по умолчанию),
  - 23**h**, 0AD**h** – целые шестнадцатеричные числа (если шестнадцатеричная константа начинается с буквы, то перед ней указывается 0),
  - 0111010**b** – целое двоичное число;
- вещественные константы формата

  [Знак] Целое [. [Целое]] [E|e [Знак] Целое],

  например: -2.1, 34**E**-28;
- символы в кодировке ASCII (MS DOS) или ANSI (Windows) в апострофах «'» или кавычках «″», например: 'A' или ″A″, при этом использование апострофов и кавычек в ассемблере эквивалентно;
- 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
        DB    ?      ; зарезервировать 1 байт памяти, доступ по адресу a+1
        DW    1234H  ; записать в слово шестнадцатеричное число 1234, доступ по
```

адресу a+2

Примечание – При записи данных размером более 1 байта в память младший байт записывается в поле с меньшим адресом, затем следует байт перед младшим и т.д. до старшего. Например, в предыдущем примере, если запись выполнялась по адресу 100, то по адресу 100 будет записано 34H, а по адресу 101 – 12H.

```
        DB      31 dup (1,2,3,4,5) ; определить 31 байт: 1, 2, 3, 4, 5, 1, 2, 3,...
val1    BYTE    255 ; записать в байт число 255₁₀ = 11111111₂ и назвать val1
```

Оглавление

```
lue3    SWORD   -128  ; записать в слово число -128 и назвать lue3
alu     BYTE    10 dup ?    ; зарезервировать 10 байт и назвать alu
        BYTE    10h    ; записать в байт шестнадцатеричное число 10₁₆ = 16₁₀
v5      BYTE    100101B  ; записать в байт двоичное число  и назвать v5
        BYTE    23,23h,0ch  ; записать в байт каждое из указанных чисел
sdk     BYTE    "Hello",0  ; записать в память строку, потом 0 и назвать sdk
        SWORD   -32767    ; записать в слово заданное число
        DWORD   12345678h  ; записать в двойное слово шестнадцатеричное число
```

Оглавление

2.3 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 инициализации данных в памяти (см. раздел 2.2).

Если операнды команд ассемблера находятся в регистрах, то в команде на соответствующих местах указываются 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 prt 28h ;
```

увеличить на 1 слово со смещением 28h

Примечание – В этом случае адресация данных выполняется по схеме:

EBP + <смещение, заданное в команде>,

но 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

(см. ниже) и непосредственное смещение, которое может задаваться числом или символическим именем.

Адрес операнда (исполнительный) 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		

Примеры:

```
inc   word ptr[500]          ; непосредственный адрес
mov   ES:[ECX],EDX           ; задана только база
mov   EAX, TABLE[ESI*4]      ; заданы индекс и масштаб
```

Косвенный адрес заключается в квадратные скобки весь или частично, например:

[OPND +ESI] OPND [ESI] OPND + [ESI] [OPND] +[ESI]

Приведенные выше формы записи косвенного адреса интерпретируются одинаково.

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.

Примеры: `[a + EBX]` и `[EBP +ESI +6]`.

В первом случае исполнительный адрес операнда определяется суммой содержимого регистра **EBX** и непосредственного смещения, заданного символическим именем «**a**», а во втором – суммой содержимого регистров **EBP**, **ESI** и непосредственного смещения, равного 6.

Примечание. При использовании косвенной адресации по схеме EBP + <смещение, заданное в команде> смещение не может быть опущено, так как частный случай адресации по данной 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 ptr** (1 байт), **word ptr** (2 байта) и **dword ptr** (4 байта), 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 ptr x, 255 ;  нас интересует только первый байт слова
       . . .
x       DW    25
```

Оглавление

2.4 Команды пересылки / преобразования данных

При описании команд ассемблера использованы следующие условные обозначения:

r8 – один из 8-ми разрядных регистров: AL, AH, BL, BH, CL, CH, DL, DH;

r16 – один из 16-ти разрядных регистров: AX, BX, CX, DX, SI, DI, SP, BP;

r32 – 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;

imm8 – непосредственно заданное 8-ми разрядное значение;

imm16 – непосредственно заданное 16-ти разрядное значение;

imm32 – непосредственно заданное 32-х разрядное значение;

imm – непосредственно заданное значение любого размера;

r/m8 – 8-ми разрядный операнд в регистре или в памяти;

r/m16 – 16-ти разрядный операнд в регистре или в памяти;

r/m32 – 32-ти разрядный операнд в регистре или в памяти;

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

rel8, rel16, rel32 – 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 (см. рисунок 11):

```
MOV Приемник, Источник
```

Допустимые варианты:

```
mov   reg, reg
mov   mem, reg
mov   reg, mem
mov   mem, imm
mov   reg, imm
mov   r/m16, sreg
mov   sreg, r/m16
```

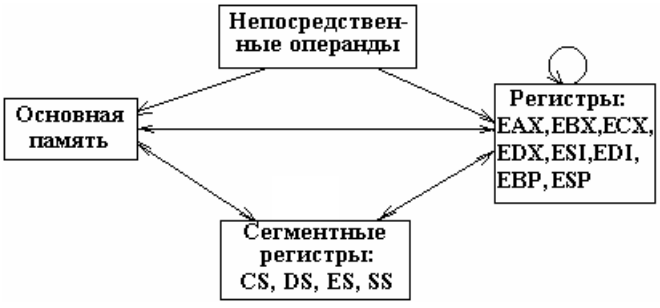


Рисунок 11 – Возможные пересылки командой MOV

Оглавление

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 r16/r32, r/m8

movzx r32, r/m16

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 / imm32 / r16 / r32 / m16 / m32  ; запись
POP    r16 / r32 / m16 / m32                  ; извлечение
```

Оглавление

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
```

Оглавление

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 ptr 8[EBX,EDI]
```

16-17. Команды умножения

В команде записывается 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 (см. ниже).

```
MUL   <Операнд2>
IMUL  <Операнд2>
```

Допустимые варианты:

mul/imul r|m8 *; AX= AL*<Операнд2>*

mul/imul r|m16 *; DX:AX= AX*<Операнд2>*

mul/imul r|m32 *; EDX:EAX= EAX*<Операнд2>*

В качестве второго операнда нельзя указать непосредственное значение!!!

Пример:

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

Оглавление

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>
IDIV    <Операнд2>
```

Допустимые варианты:

```
div/idiv  r|m8    ;  AL= AX:<Операнд2>, AH – остаток
div/idiv  r|m16   ;  AX= (DX:AX):<Операнд2>, DX – остаток
div/idiv  r|m32   ;  EAX= (EDX:EAX):<Операнд2>, EDX – остаток
```

В качестве второго операнда нельзя указать непосредственное значение!!!

Пример:

```
mov    AX,40       ; загрузка делимого
```

Оглавление

```
cwd                 ; развертывание делимого до 4-х байт в DX:AX
idiv  word ptr 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
          .DATA?
X         SWORD ?
```

Примечание. Использование сегмента неинициализированных данных не является обязательным. Все переменные можно описать в сегменте инициализированных данных.

В сегменте кода записываем фрагмент вычисляющей программы:

```
          .CODE
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
          . . .
```

Оглавление

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**.**

Оглавление

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

far ptr – 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 указатель **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 ptr**, что и подразумевается по умолчанию.

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

Оглавление

- <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»
```

Оглавление

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»).

Все команды имеют однобайтовое поле адреса (формат short), следовательно смещение имени перехода относительно команды не должно превышать -128...127 байт. 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.

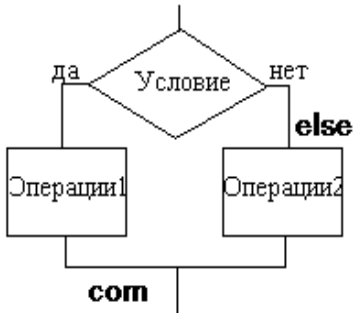
Оглавление

3.2.1 Программирование ветвлений

Ветвления программируются с использованием команд условной и безусловной передачи управления.

В начале 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)**:

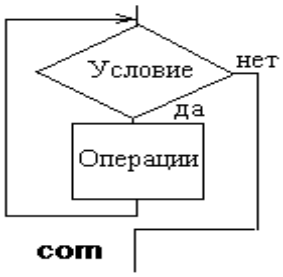
```
        mov     ax,A
        cmp     ax,B      ; сравнение A и B
        jl      LESS      ; переход по меньше
        mov     X,ax
        jmp     CONTINUE  ; переход на конец ветвления
LESS:   mov     ax, B
        mov     X,ax
CONTINUE: …
```

Оглавление

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
```

Оглавление

3.3 Команды организации циклической обработки. Организация счетных циклов

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

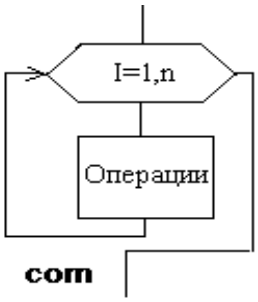
1. **Команда организации счетного цикла:**

LOOP Адрес перехода

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:  ...
```

Оглавление

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  EAX,[500] или lea  EAX,500  ; в EAX загружается число 500
lea  EDX,[ECX]                   ; в EDI загружается число из ECX
lea  EBX,TABLE[ESI*4]            ; в EBX загружается  число из ESI, умноженное на 4
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 определении адресов массивов, матриц и строк.

Оглавление

3.4.1 Обработка одномерных массивов

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

```
A       SWORD   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,MAS[EBX*2]
        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≤10).

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

Оглавление

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 Паскале 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,A[EBX*2+2]    ;  сравниваем числа
        jge   NEXT             ;  если больше, то перейти к следующему
        mov   AX,A[EBX*2+2]    ; 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
        mov   DX,A[EBX]        ; заносим первый элемент столбца
```

Оглавление

```
        mov   ESI,10              ; смещение второго элемента столбца
CYCL2:  cmp   DX,A[EBX]+[ESI]     ; сравниваем
        jge   NEXT                ; если больше или равно - к следующему
        mov   DX,A[EBX]+[ESI]     ; 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.

Оглавление

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).

Оглавление

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:SI,** 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.

Оглавление

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**.**

Оглавление

4 Более сложные машинные команды ассемблера

4.1 Команды манипулирования битами

1. Логические команды.

NOT Операнд ; логическое НЕ;

AND Операнд1, Операнд2 ; логическое И;

OR Операнд1, Операнд2 ; логическое ИЛИ;

XOR Операнд1, Операнд2 ; исключающее ИЛИ;

TEST Операнд1, Операнд2 ; И без записи результата.

Операнды байты или слова.

Пример. Выделить из числа в AL первый бит:

```
and    al,10000000B
```

2. Команды сдвига

Код операции Операнд, Счетчик

Счетчик записывается в регистр CL. Если счетчик равен 1, то его можно записать в команду.

Коды команд сдвига:

SAL, SHL – сдвиг влево арифметический и логический;

SAR, SHR – сдвиг вправо арифметический и логический;

ROL, ROR – сдвиг влево и вправо циклический;

RCL, RCR– сдвиг циклический влево и вправо с флагом переноса;

Пример. Умножить число в AX на 10:

```
mov    bx,ax
shl    ax,1
shl    ax,1
add    ax,bx
shl    ax,1
```

Оглавление

4.2 Организация ввода – вывода в консольном режиме

Библиотека MASM32.lib содержит специальные подпрограммы организации ввода-вывода для консольного режима.

Ввод. Процедура ввода:

StdIn PROC lpszBuffer:DWORD, bLen:DWORD

Первый операнд – адрес буфера ввода, второй – размер буфера ввода (до 128 байт).

При вызове процедуры компьютер переходит в состояние ожидания ввода с клавиатуры. Ввод завершается при нажатии клавиши «Enter». В буфере ввода после завершения операции находятся коды введенных символов. Строка завершается маркером конца строки (0Dh,0Ah), например, если пользователь ввел символы «+ 123» (между плюсом и 1 введен один пробел), то содержимое буфера в шестнадцатеричном виде будет следующим:

2B 20 31 32 33 0D 0A , где $2B_{16}$ – код ANSI символа «+», а 20_{16} – код пробела.

Таким образом, если пользователь ввел пробел, знак, буквы или цифры, то в буфере будут находиться их шестнадцатеричные коды.

Таким образом, при программировании операций ввода на ассемблере приходится осуществлять преобразования чисел из символьного представления во внутренний формат. Для целых чисел – это двоичный формат с фиксированной точкой, согласно которому отрицательные числа должны быть записаны в дополнительном коде.

Для облегчения преобразования во внутренний формат целесообразно оговорить возможные варианты ввода чисел в символьном виде, например, может или не может быть введен знак, возможны ли пробелы перед числом и т. д.

Алгоритм преобразования основан на схеме Горнера:

<число>:=<число>*10+<цифра>.

Алгоритм преобразования для положительных чисел без знака следующий**:**

Число:=0

Ввести Код цифры

Цикл-пока Код цифры≠0Dh

Цифра:=Код цифры – 30h

Оглавление

Число:=Число*10 + Цифра

Ввести Код цифры

Все-цикл

Если число со знаком или формат ввода допускает наличие пробелов, то это должно обрабатываться отдельно.

Библиотеки ассемблера содержать специальные подпрограммы преобразования, которые также могут быть использованы.

Процедура замены маркера конца строки (0Dh,0Ah) нулем:

StripLF PROC lpszBuffer:DWORD

Параметр – адрес буфера ввода. После выполнения процедуры введенная строка будет завершаться нулем, например, для примера, приведенного выше, это будет:

2B 20 31 32 33 **00** 0A .

Что позволит для преобразования числа использовать стандартную функцию из библиотеки C++.

Функция преобразования завершающейся нулем строки в число:

atol PROC lpszBuffer:DWORD ; результат – в EAX

Для вызова функции необходимо, чтобы:

1) был подключен файл, содержащий описание прототипа этой функции, для стандартных функций это, как правило, файлы **kernel32.inc** и **masm32.inc**:

```
Include kernel32.inc
Include masm32.inc
```

2) был подключен файл библиотеки, содержащий оттранслированный текст функции, для стандартных функций это, как правило, файлы **kernel32.lib** и **masm32.lib**:

```
IncludeLib kernel32.lib
IncludeLib masm32.lib
```

Вызов осуществляется макрокомандой Invoke:

```
INVOKE Имя процедуры или ее адрес [, Список аргументов]
```

При указании аргументов часто используются атрибуты полей данных:

ADDR <Имя поля данных**>** – возвращает ближний или дальний адрес переменной в зависимости от модели памяти – для Flat ближний;

Оглавление

OFFSET <Имя поля данных> – возвращает смещение переменной относительно начала сегмента – для Flat совпадает с ADDR;

TYPE <Имя поля данных> – возвращает размер в байтах элемента описанных данных;

LENGTHOF <Имя поля данных> – возвращает количество элементов, заданных при определении данных;

SIZEOF <Имя поля данных> – возвращает размер поля данных в байтах.

Пример. Программа ввода числа

```
          .DATA
zapros    DB    'Input value:',13,10,0 ;  запрос
buffer    DB    10 dup ('0')      ;  буфер ввода

          .CODE
          . . .
vvod:     Invoke StdOut,ADDR zapros
          Invoke StdIn,ADDR buffer,LengthOf buffer
          Invoke StripLF,ADDR buffer
;  Преобразование в SDWORD
          Invoke atol,ADDR buffer ;результат в EAX
          . . .
```

Вывод. При выводе решается обратная задача: необходимо преобразовать число, представленное во внутреннем формате, в символьную строку, завершающуюся нулем. Обратное преобразование из внутреннего формата в символьный обычно использует стандартное правило перевода числа из двоичной системы счисления в десятичную: деление на 10 с выделением остатков. В этом случае десятичные цифры получаются в обратном порядке. Если среди выводимых чисел могут быть отрицательные, то необходимо предусмотреть специальную проверку и преобразовывать отрицательные числа в прямой код.

Гораздо проще для этого применить стандартную процедуру.

Процедура преобразования числа в строку:

```
dwtoa     PROC     public dwValue:DWORD, lpBuffer:PTR BYTE
```

Оглавление

Первый параметр – целое число формата DWORD, второй – адрес буфера, размером 16 байт. Полученная после преобразования строка обычно короче 16 символов, так как не содержит незначащих нулей.

Процедура вывода *завершающейся нулем* строки в окно консоли:

StdOut PROC lpszBuffer:DWORD

Параметр – адрес буфера вывода.

Пример. Программа вывода числа

```
        .DATA
result  DWORD ?          ; поле результата
string  DB    13,10,'Result =' ; заголовок вывода
resstr  DB    16 dup (?)      ; выводимое число-строка
        .CODE ...
; Преобразование числа в символьную строку
        Invoke dwtoa,result,ADDR resstr
; Вывод заголовка и результата-строки
        Invoke StdOut,ADDR string
        . . .
```

Пример. Программа ввода и вывода элементов массива с преобразованием во внутреннее представление

Написать программу ввода массива из **10** чисел размером двойное слово и вывода того же массива. Числа должны вводиться каждое в своей строке. Вывод всех чисел должен осуществляться в одну строку через пробелы. Перед отрицательными числами необходимо выводить знак "-".

```
.586 ; разрешает использование набора команд i80586
.MODEL  flat, stdcall ; определяет модель памяти и тип связи
 OPTION CASEMAP:NONE ; чувствительность идентификаторов к регистру
Include kernel32.inc ; подключает файлы описаний библиотечных
Include masm32.inc   ; процедур и функций
IncludeLib kernel32.lib ; подключает библиотеки
IncludeLib masm32.lib   ; на этапе компоновки
```

Оглавление

```
        .DATA       ; сегмент инициализированных данных
Msg      DB   "Press Enter to Exit",0AH,0DH,0
InputMsg DB   "Input integer value",0AH,0DH,0
OutputMsg DB  "Results:",0AH,0DH,0
pusto    DB   " ",0
nl       DB   0AH,0DH,0

        .DATA?      ; сегмент неинициализированных данных
inbuf    DB     100 DUP (?)
a        SDWORD 10 DUP (?)
string   DB     16 DUP (?)
        .STACK  4096  ; сегмент стека - 4096 байт
        .CODE       ; сегмент кода
Start:   mov    ECX,10
         mov    EBX,0
cycle:   push   ECX
         Invoke StdOut,ADDR InputMsg ; вывод запроса
         Invoke StdIn,ADDR inbuf,LengthOf inbuf ; ввод числа
         Invoke StripLF,ADDR inbuf
         Invoke atol,ADDR inbuf
         mov    a[EBX*4],EAX
         inc    EBX
         pop    ECX
         loop   cycle
         Invoke StdOut,ADDR OutputMsg ; вызов процедуры вывода
         mov    ECX,10
         mov    EBX,0
cycle2:  push   ECX
         Invoke dwtoa,a[EBX*4],ADDR string
                             ; преобразование числа в символьную строку
         Invoke StdOut,ADDR string ; вывод результата-строки
         Invoke StdOut,ADDR pusto ; вывод пробела
```

Оглавление

```
inc     EBX
pop     ECX
loop    cycle2
Invoke StdOut,ADDR nl ;  вывод маркера конца строки
Invoke StdOut,ADDR Msg ;  вызов процедуры вывода
Invoke StdIn,ADDR inbuf,LengthOf inbuf
                                    ; вызов процедуры ввода
Invoke ExitProcess,0    ; вызов процедуры завершения
END     Start
```

Оглавление

Контрольные вопросы

1. Какие команды относятся к командам манипулирующим с битами?

Ответ.

2. Как реализованы команды ввода-вывода?

Ответ.

Оглавление

Литература

1. Ирвин К. Язык ассемблера для процессоров Intel. – М.: Изд. дом «Вильямс», 2005.
2. Зубков С.В. Assembler для DOS, Windows и Unix. – М.: ДМК Пресс, 2004.
3. Пирогов В.Ю. Ассемблер. Учебный курс. – СПб.: БХВ-Петербург, 2003.
4. Финогенов К.Г., Рудаков П.И. Язык Ассемблера: уроки программирования. – М.: Диалог-МИФИ, 2001.
5. Юров В.И. Справочник по языку Ассемблера IBM PC. – СПб.: Питер, 2004.

Оглавление