	<p>Харьковский национальный университет радиоэлектроники Кафедра ЭВМ ТОРБА</p>	<p>Украина, 61166, г.Харьков, пр. Ленина 14, ауд. 221</p>
	<p>Александр Алексеевич Кандидат технических наук, профессор</p>	<p>Раб.тел: (8-057) 702-13-54 Email: alexandr.torba@nure.ua</p>

КОНСПЕКТ ЛЕКЦИЙ **И** **МЕТОДИЧЕСКИЕ УКАЗАНИЯ** к лабораторной работе по программированию **ЦЕЛОЧИСЛЕННОГО ПРОЦЕССОРА** **x86 (C P U)**

Утверждено на заседании каф. ЭВМ
 Протокол №__ от «__» ____ 2020

Зав. кафедрой ЭВМ,

проф.

Коваленко А.А.

2021

1 АРХИТЕКТУРА 32-х РАЗРЯДНЫХ МИКРОПРОЦЕССОРОВ 386+

Базовая архитектура 32-х разрядных процессоров (обозначаемых: 386+) является общей для существующих на данный момент процессоров фирмы INTEL – 386, 486, PENTIUM и его модификаций.

МП состоит из 3 основных частей:

- устройства обработки;
- устройства управления ЗУ;
- интерфейсного блока.

УСТРОЙСТВО ОБРАБОТКИ состоит из исполнительного устройства (операционной части) и блока команд (управляющей части). Содержит 8 32-х разрядных РОН, 64-х битовый циклический сдвигатель. Умножение и деление осуществляется на 1 бит за цикл. Алгоритм умножения такой, что процесс прекращается, когда наиболее значащий бит, умножается на все нули. Типичное время умножения 32-х разрядных чисел около 1 мкс (для процессора I80386).

УСТРОЙСТВО УПРАВЛЕНИЯ ЗУ состоит из сегментного и страничного блоков. Сегментный блок позволяет работать с логическими адресами со всеми вытекающими отсюда преимуществами. Страничная организация используется внутри сегмента и управляет физическими адресами. Каждая задача может иметь до 16381 (2^{14}) сегмента до 4 Гбайт каждый (2^{32}), т.е. виртуальная память может быть размером 64 Тбайт (2^{46}).

ИНТЕРФЕЙСНЫЙ БЛОК обеспечивает взаимодействие с внешними устройствами, включая автоматическое управление разрядностью шины, и формирование сигналов активности байтов.

МП 386+ могут функционировать в трех режимах:

- **REAL ADDRESS MODE** – режим реальной адресации (PPA) – характеризуется тем, что МП работает как очень быстрый 8086 с 32-битовым расширением; в этом режиме возможна адресация 1 Мбайта физической памяти (на самом деле, как у I80286, - почти на 64 Кбайта больше);
- **PROTECTED ADDRESS MODE** – режим защищенной виртуальной адресации (PBA) – реализует все достоинства МП (режим параллельного выполнения нескольких задач несколькими 8086 – по одному на задачу). На одном процессоре в таком режиме могут одновременно исполняться несколько задач с изолированными друг от друга реальными ресурсами. При этом использование физического адресного пространства памяти управляется механизмами сегментации и трансляции страниц. Попытки выполнения недопустимых команд, выхода за рамки отведенного пространства памяти и разрешенной области ввода-вывода контролируются системой защиты.
- **VIRTUAL 8086 MODE** – режим виртуального процессора 8086 (сокращенно – V86). Прикладная программа, которая выполняется в этом режиме, полагает, что она работает на процессоре 8086. Однако, некоторые команды, в основном связанные с управлением вводом-выводом, программе выполнять запрещается, поэтому при нарушении защиты генерируется прерывание и управление передается операционной системе.

1 ПРОГРАММНАЯ МОДЕЛЬ 32-х РАЗРЯДНЫХ ПРОЦЕССОРОВ (386+)

МП 386+ имеет 31 регистр (у PENTIUM+ – 32 регистра), разбитые на следующие группы:

- регистры общего назначения;
- сегментные регистры;
- указатель команд и регистр флагов (признаков);

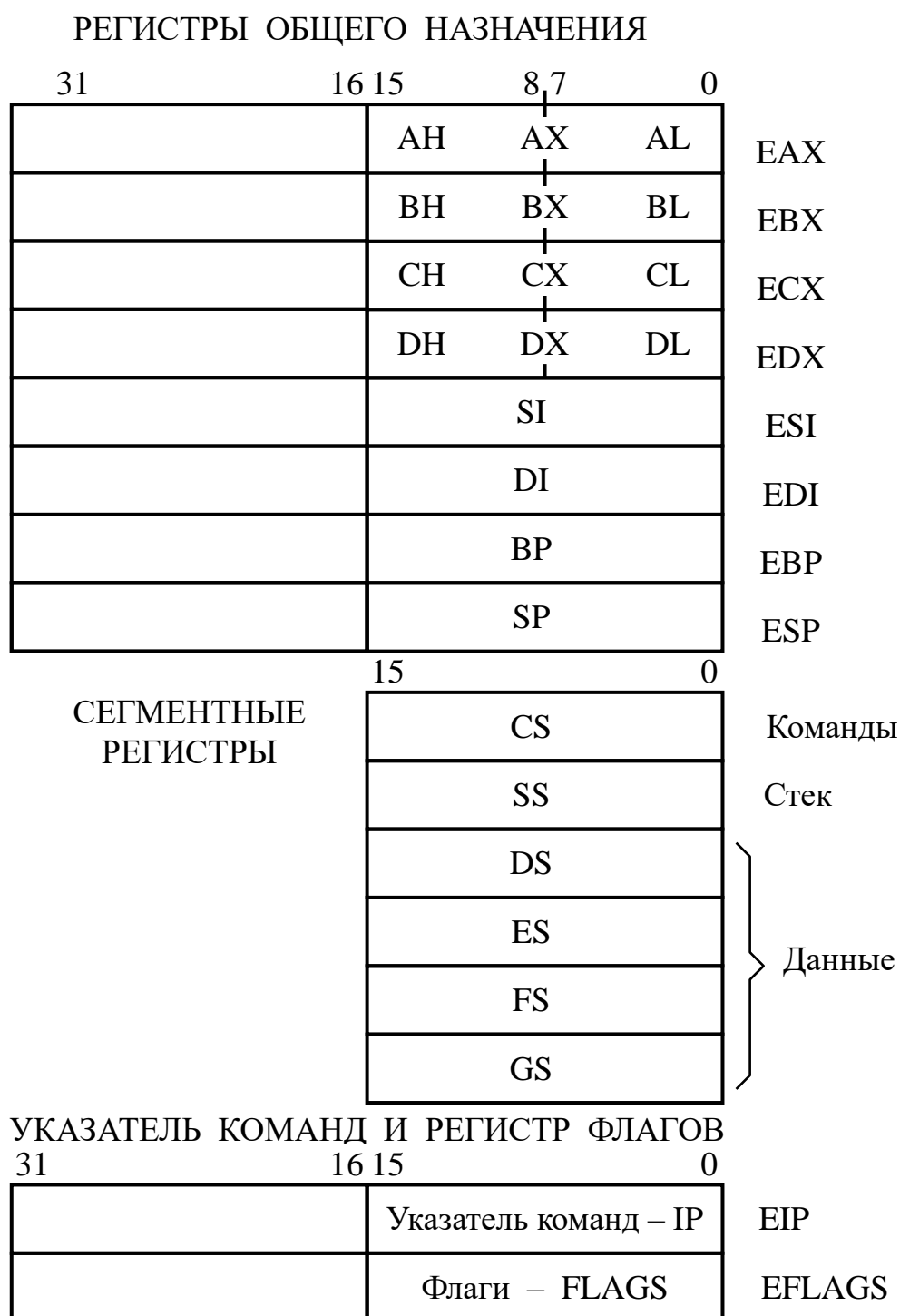


Рис. 1 – Регистры 32-х разрядных МП (386+)

- управляющие регистры;
- регистры системных адресов;
- отладочные регистры;
- тестовые регистры.

Набор **РЕГИСТРОВ ОБЩЕГО НАЗНАЧЕНИЯ** (рис. 1) включает соответствующие регистры процессоров I8086 и I80286. Все эти регистры, кроме сегментных, имеют разрядность 32 бита и к прежнему обозначению их имен добавилась приставка «Е» (Extended – расширенный). Отсутствие приставки «Е» в имени означает ссылку на младшие 16 бит расширенных регистров. Обратиться к старшим 16-ти битам расширенных регистров ни одна команда не может. Как и в I8086, возможно независимое обращение к младшему и старшему байтам регистров AX, BX, CX, DX.

Архитектура МП 386+ позволяет непосредственно обращаться к 6 сегментам (размером до 4 Гбайт каждый) при помощи специальных селекторов, которые загружаются в сегментные регистры программно. Содержимое РОНов, селекторов, указателя команд и регистра флагов (признаков) зависит от выполняемой задачи и автоматически перегружается при переключении задач.

Остальные регистры МП используются, главным образом, для упрощения проектирования и отладки операционной системы.

Регистры общего назначения (РОН) – используются для хранения операндов и адресов. Могут работать с операндами, имеющими длину 1, 8, 16, 32 и 64 бита или с битовыми полями длиной от 1 до 32 бит.

УКАЗАТЕЛЬ КОМАНД – EIP – хранит смещение, которое всегда складывается со значением кодового сегментного регистра (CS) и определяет адрес следующей команды. При 16-ти битовой адресации используются только младшие 16 бит (IP).

РЕГИСТР ФЛАГОВ (признаков) – EFLAGS (рис. 2) – отражает состояние МП. При использовании только 16-ти младших разрядов - регистр флагов совместим с предыдущими моделями МП (см. рис. 1.3 и рис. 1.4).

На рис. 2 обозначены символами:

x – системный флаг;

s – флаг состояния;

c – управляющий флаг.

31										16 15										0										
0	0	0	0	0	0	0	0	0	0	I D	V I P	V I F	A C	V M	R F	0	N T	IO PL	OF	DF	IF	TF	SF	ZF	0	A F	0	P F	1	C F
										x	x	x	x	x	x		x	x	x	x	c	x	x	s	s		s		s	s

Рис. 2 - Регистр флагов EFLAGS (признаков)

НАЗНАЧЕНИЕ БИТ РЕГИСТРА ФЛАГОВ (ПРИЗНАКОВ)

CF - (Carry Flag) – флаг переноса, показывающий перенос (заем) из старшего бита при арифметических операциях, а также значение выдвигаемого бита при сдвиге операнда;

AF - (Auxiliary Flag) – флаг вспомогательного переноса (заема) в младшей тетраде для десятичной арифметики;

OF - (Overflow Flag) – флаг арифметического переполнения, определяющий (при OF=1) выход знакового результата за границы диапазона;

ZF - (Zero Flag) – флаг нуля, показывающий (при ZF=1) нулевой результат команды;

SF - (Sign Flag) – флаг знака, дублирует значение старшего бита результата, который при использовании дополнительного кода соответствует знаку числа;

PF - (Parity Flag) – флаг паритета (четности), фиксирующий (при PF=1) наличие в младшем байте результата четного числа единичных бит.

IOPL (Input/Output Privilege Level) - используется только в РВА. IOPL указывает максимальную величину текущего приоритета, обеспечивающую выполнение команд ввода-вывода (В – В) без реакции на 13 ошибку. Этот признак также обеспечивает выбор IF, когда новое значение выталкивается из стека в регистр признаков. POPF и IRET могут изменять IOPL поле, когда IOPL = 0 (CPL=0). При переключении задач IOPL может изменяться всегда при переписи TSS (286+).

NT - (Nested Task Flag) – флаг вложенной задачи (286+);

ID (Id Flag) – флаг доступности команды идентификации CPUID (PENTIUM+ и некоторые 486+);

VIP (Virtual Interrupt Pending) – виртуальный запрос прерывания (PENTIUM+);

VIF (Virtual Interrupt Flag) – виртуальная версия флага IF (разрешения прерывания) для многозадачных систем (PENTIUM+).

AC (Alignment Check) – флаг контроля выравнивания. При исполнении программ на уровне привилегий 3 в случае обращения к операнду, не выровненному по соответствующей границе (2,4,8 байт), и при установленном флаге AC произойдет исключение-отказ 17 с нулевым кодом ошибки. На уровнях привилегий 0,1,2 контроль выравнивания не производится (486+).

VM (Virtual 8086 Mode) – обеспечивает режим виртуального 8086 внутри режима виртуальной адресации. При VM = 1 МП будет переключен в режим виртуального I8086, при этом управление перезагрузкой сегментов будет осуществляться подобно I8086, но с исключением 13 недействительных привилегированных команд. VM может быть установлен в РВА командой IRET (если уровень приоритета = 0) и задача переключается на более низший уровень. Команда POPF не влияет на VM. Команда PUSHF всегда сбрасывает VM в 0, если она выполняется в режиме виртуального 8086. Содержимое регистра признаков будет копироваться при прерываниях

или сохраняться при переключении задачи, если прерывание будет выполняться в режиме виртуального 8086 (386+).

RF (Resume Flag) – флаг возобновления – используется совместно с отладочными регистрами контрольных точек (прерываний) или пошагового режима. С его помощью проверяется ход выполнения команд в отладочном режиме (процесс отладки). Если установлен $RF = 1$, то это позволяет игнорировать ошибки, возникающие при отладке до следующей команды. RF автоматически сбрасывается в 0 при успешном выполнении команды (ошибки не обнаружены), за исключением команд IRET и POPF, а также JMP, CALL и INT при переключении задач. Эти команды устанавливают RF в состояние, определяемое состоянием памяти. Например, в конце выполнения подпрограммы обслуживания контрольной точки команда IRET может установить RF в состояние, соответствующее значению регистра признаков, хранимого в стеке без повторной установки RF в 1 (386+).

NT (Nested Task Flag) – флаг вложенной задачи (гнездования) используется только в режиме виртуального адреса (РВА). $NT=1$ указывает, что текущая задача является вложенной по отношению к другой задаче. Этот бит устанавливается и сбрасывается при вызове других задач. NT проверяется командой IRET для определения внутри заданного или внешнего по отношению к данной задаче возврата. Команды POPF и IRET будут устанавливать NT в соответствии с тем, что хранится в стеке для любого уровня привилегированности (286+).

МП 386+ содержат 6 16-ти битовых **СЕКМЕНТНЫХ РЕГИСТРОВ** (у предыдущих поколений – только 4 сегментных регистра), хранящих значение селектора и определяющих значения начальных (базовых) адресов сегментов. В РВА каждый сегмент может изменяться в диапазоне от одного байта до максимального значения физического адресного пространства 4 Гбайта. В РРА размеры сегмента ограничены размером 64 Кбайт.

ДЕСКРИПТОРНЫЕ РЕГИСТРЫ СЕКМЕНТОВ программно не видимы, но они неразрывно связаны с соответствующими сегментными регистрами (рис. 3). Каждый дескрипторный регистр хранит 32-х битовый базовый адрес сегмента, 20-ти битовый размер сегмента и другие необходимые атрибуты сегмента.

Когда значение селектора загружается в сегментный регистр, **в режиме виртуальной адресации (РВА)** соответствующий дескрипторный регистр автоматически загружается информацией из дескрипторной таблицы.

В РВА базовый адрес, размер и атрибуты сегментного дескриптора определяется селектором. 32-х битовый **БАЗОВЫЙ АДРЕС** сегмента становится компонентом формирования исполнительного адреса, 20-ти битовый **РАЗМЕР СЕКМЕНТА** используется для проверки границ рабочей области, а **АТРИБУТЫ** проверяются на соответствие типу запрашиваемой памяти (типу обращения).

В РРА непосредственно используется только базовый адрес (со сдвигом на 4 разряда влево), а размеры сегмента и атрибуты постоянны

(фиксированы для РРА).

Сегментные регистры		Дескрипторные регистры – программно недоступны (загружаются автоматически)		
		Базовые адреса сегментов	Размеры сегментов	Атрибуты сегментов
15	0			
Селектор	CS			
Селектор	SS			
Селектор	DS			
Селектор	ES			
Селектор	FS			
Селектор	GS			

Рис. 3 – Сегментные регистры и соответствующие дескрипторные регистры МП 386+

2 ТИПЫ ДАННЫХ 32-х БИТОВЫХ ПРОЦЕССОРОВ 386+

32-х разрядные процессоры фирмы INTEL (386+) работают с целыми двоичными числами длиной 8, 16 или 32 бита и двоично-кодированными десятичными числами (BCD-числами) длиной 8 бит. Двоичные числа допускают интерпретацию как целых без знака и целых со знаком, а десятичные (BCD) – знака не имеют.

В ДВОИЧНЫХ ЦЕЛЫХ ЧИСЛАХ БЕЗ ЗНАКА все разряды считаются значащими (см. рис. 4). ДВОИЧНЫЕ ЦЕЛЫЕ ЧИСЛА СО ЗНАКОМ представляются в дополнительном коде. Старший бит является знаковым (рис. 4): $S = 0$ – число положительное, $S = 1$ – число отрицательное.

ДЕСЯТИЧНЫЕ ЧИСЛА представляются в упакованном и неупакованном форматах. Упакованный формат предполагает, что байт содержит две десятичные цифры в коде с весами 8421, занимающих младшую и старшую тетрады. Диапазон представимых BCD-чисел – 0...99 (рис. 2.4). В неупакованном формате байт содержит одну десятичную цифру, которая обычно изображается в символьном коде ASCII.

Новые команды процессоров 386+ поддерживают БИТОВЫЕ ДАННЫЕ:

- БИТ – одиночный двоичный разряд.
- БИТОВОЕ ПОЛЕ – группа до 32-х битов.
- ЦЕПОЧКА БИТОВ (СТРОКА) – набор последовательных битов, длиной до 4 Гбит.

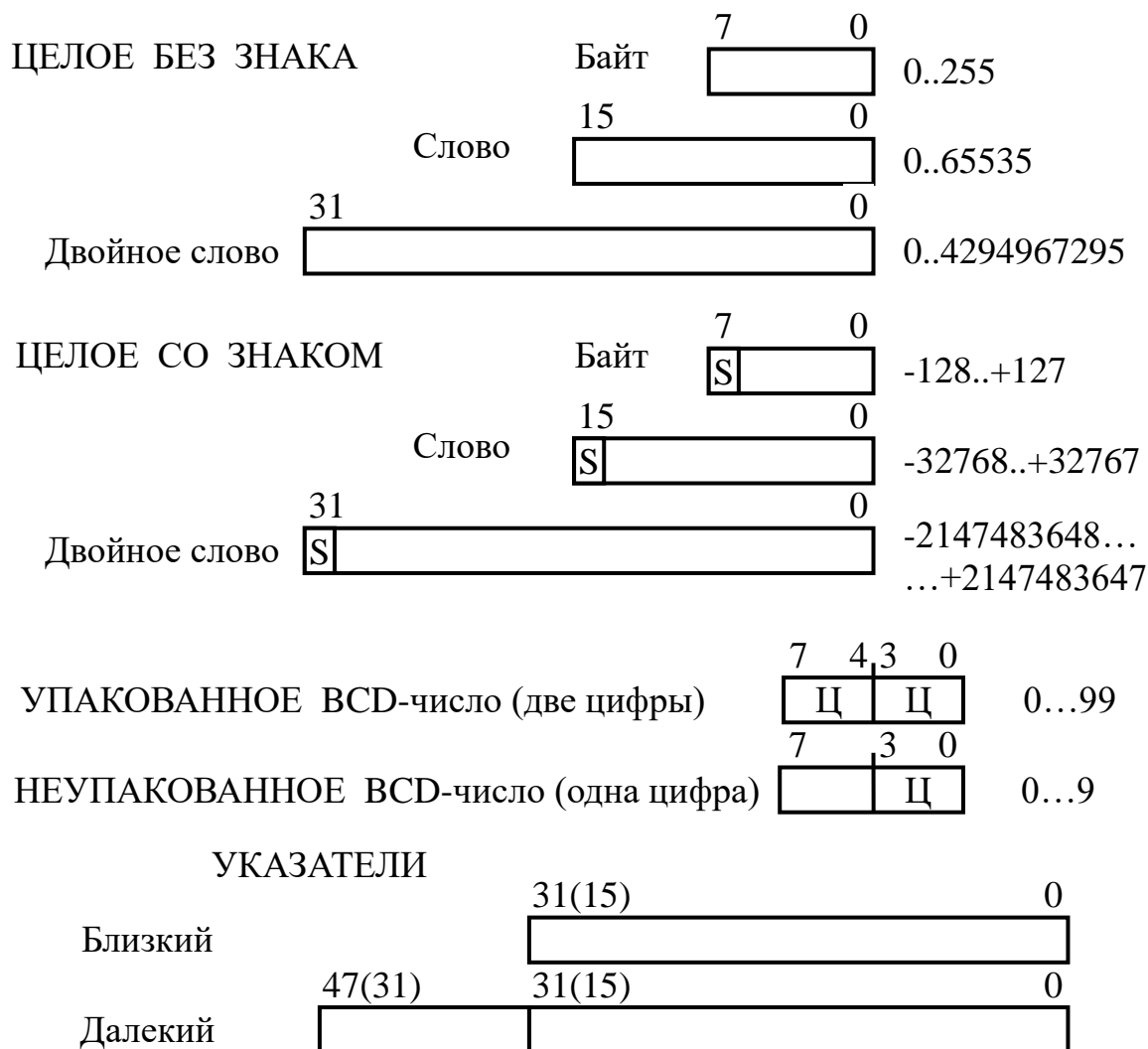


Рис. 4 – Типы данных 32-х разрядных процессоров (386+)

Процессор может легко оперировать с цепочками бит, байт, слов и двойных слов. Под ЦЕПОЧКОЙ (string) понимается последовательность практически любой длины отдельных, но взаимосвязанных элементов данных, ХРАНЯЩИХСЯ ПО СОСЕДНИМ АДРЕСАМ.

УКАЗАТЕЛИ применяются для обращения к некоторым объектам в памяти, например, адресам подпрограмм. Близкие (NEAR) или внутрисегментный указатель (см. рис. 4) – это 16-ти битовое или 32-х битовое смещение внутри текущего сегмента. Далекий (FAR) или межсегментный указатель применяется в тех случаях, когда программа осуществляет передачу управления в другой сегмент. Такой указатель определяет новый сегмент (с помощью селектора) и 16-ти или 32-х битовое смещение внутри этого сегмента.

При размещении операндов в памяти необходимо учитывать, что процессоры 386+ не накладывают ограничения на размещение данных. Однако производительность процессора повышается, если слова размещены по четным адресам, а двойные слова – по адресам, кратным четырем. Такой

принцип называется **ВЫРАВНИВАНИЕ АДРЕСОВ** по границам слов или двойных слов. Выравнивание особенно важно для стека, который работает только со словами или двойными словами.

3 СИСТЕМА КОМАНД ПРОЦЕССОРОВ 386+

Система команд включает 9 групп команд:

1. передачи данных;
2. арифметические и логические;
3. сдвига;
4. обработки строк;
5. манипуляции битами;
6. передачи управления;
7. поддержки языков высокого уровня;
8. поддержки операционной системы;
9. управления процессором.

Команды могут содержать от 0 до 3 операндов, размещенных в регистрах, памяти или непосредственно в команде. Большинство безоперандных команд – однобайтовые. Однооперандные команды обычно – двухбайтовые. Средняя длина команды – 3,2 байта. Это позволяет хранить в среднем 5 команд в 16-ти байтовой **ОЧЕРЕДИ КОМАНД БЛОКА ОПЕРЕЖАЮЩЕЙ ВЫБОРКИ**.

При использовании двух операндов возможны следующие типы взаимодействия:

- регистр - регистр;
- память - регистр;
- регистр - память;
- непосредственный операнд - регистр;
- непосредственный операнд - память;
- память - память.

Операнды могут быть 8, 16 или 32-х разрядными. Когда выполняются команды, написанные для 386+, операнды имеют длину 8 или 32 бита, когда – для 80286 и 8086 – операнды 8 или 16 бит. Ко всем инструкциям могут добавляться префиксы, которые изменяют длину операндов (т.е. позволяют использовать 32-х битовые операнды в 16-ти битовых командах или 16-ти битовые операнды в 32-х битовых командах).

3.1 РЕЖИМЫ (МЕТОДЫ) АДРЕСАЦИИ

Процессоры 386+ обеспечивает 13 режимов адресации, которые рассчитаны на эффективное выполнение программ, написанных на языках высокого уровня (ЯВУ) типа: C++, Фортран и др..

НЕЯВНАЯ АДРЕСАЦИЯ. Операнд адресуется неявно, если в команде нет специальных полей для его определения, т.е. операнд задается

полем команды. В ассемблерных кодах с неявной адресацией поле операнда пустое. Примеры команд с неявной адресацией:

AAA ; Коррекция регистра AL после сложения
CMC ; Инверсия флага переноса
STD ; Установить в 1 флаг направления.

РЕЖИМ РЕГИСТРОВОЙ АДРЕСАЦИИ и

РЕЖИМ НЕПОСРЕДСТВЕННОЙ АДРЕСАЦИИ – предназначены,

соответственно, для адресации одного из регистров регистрового блока или непосредственного операнда в команде с разрядностью 8, 16 или 32 бита :

INC esi ; Инкремент регистра ESI
SUB ECX, ECX ; Сбросить регистр ECX
MOV EAX, CR0 ; Передать в EAX содержимое CR0.
MOV EAX, 0F0F0F0F0h ; Загрузить константу в EAX
AND AL, 0FH ; Выделить младшую тетраду регистра AL
BT EDI, 3 ; Передать во флаг CF третий бит
; регистра EDI

Имеется 10 режимов АДРЕСАЦИИ ПАМЯТИ. Исполнительный адрес включает в себя два компонента адреса ячейки памяти – сегмент и эффективный адрес (внутрисегментное смещение). ЭФФЕКТИВНЫЙ АДРЕС (ЕА) вычисляется суммированием следующих элементов :

- СМЕЩЕНИЕ (отклонение) – целая 8-ми или 32-х битовая величина со знаком, непосредственно задаваемая в команде (16-ти битовые отклонения могут использоваться при помощи префикса);
- БАЗА – содержимое любых РОНов. Базовые регистры обычно используются компиляторами в качестве точки отсчета локальной области памяти;
- ИНДЕКС – содержимое любых РОНов, исключая ESP. Индексные регистры используются для доступа к элементам строк или массивов.
- МНОЖИТЕЛЬ f - указывает шаг (1, 2, 4 или 8) для индексного регистра. Шаг индексации позволяет успешно адресовать массивы или структуры, содержащие многобайтовые операнды.

$$EA = \text{БАЗА} + \text{ИНДЕКС} * (\text{ШАГ ИНДЕКСАЦИИ}) + \text{ОТКЛОНЕНИЕ}.$$

Вычисление эффективного адреса (ЕА) практически не ухудшает производительность процессора из-за использования конвейерного режима.

РЕЖИМЫ АДРЕСАЦИИ ПАМЯТИ :

ПРЯМАЯ АДРЕСАЦИЯ – смещение (отклонение) адреса операнда содержится в 8, 16 или 32 разрядах команды :

MOV AL, [2000h] ; Передать байт в регистр AL

INC dword prt [123456h] ; Инкремент двойного слова
 ; в памяти.

РЕГИСТРОВЫЙ КОСВЕННЫЙ МЕТОД АДРЕСАЦИИ – базовый или индексный регистр содержат адрес операнда :

MOV	AL, [ECX]	; Передать в AL байт по адресу из ECX
DEC	word prt [ESI]	; Декремент слова по адресу из ESI.

БАЗОВАЯ АДРЕСАЦИЯ – базовый регистр суммируется с отклонением:

MOV	EAX, [EBX+4]	; Передать двойное слово из памяти
ADD	[ECX+10h], DX	; Прибавить к слову в памяти.

ИНДЕКСНАЯ АДРЕСАЦИЯ – индексный регистр (любой РОН кроме ESP) суммируется с отклонением :

SUB	array[ESI], 2	; Вычесть 2 из элемента массива
IMUL	vector[ECX]	; Умножить EAX на элемент массива.

ИНДЕКСНАЯ АДРЕСАЦИЯ С ШАГОМ – содержимое индекс-ного регистра умножается на шаг «f» и суммируется с отклонением :

MOV	EAX, vec[ECX*4]	; Переслать в EAX двойное слово
		; из массива.

БАЗОВО-ИНДЕКСНАЯ АДРЕСАЦИЯ. – ЕА = БАЗА + ИНДЕКС :

ADD EAX, [EBX][ESI] ; Прибавить к EAX двойное
: слово из памяти.

БАЗОВО-ИНДЕКСНАЯ АДРЕСАЦИЯ С ШАГОМ. – $EA = \text{БАЗА} + \text{ИНДЕКС} * \text{ШАГ}$:

INC word prt [EDX][EDI*4] ; Інкремент ячейки пам'яті.

БАЗОВО-ИНДЕКСНАЯ АДРЕСАЦИЯ С ОТКЛОНЕНИЕМ. –
БА = БАЗА + ИНДЕКС + ОТКЛОНЕНИЕ:

MOV AX, [ECX][ESI+20h] ; Переслать слово из памяти

БАЗОВО-ИНДЕКСНАЯ АДРЕСАЦИЯ С ОТКЛОНЕНИЕМ И С ШАГОМ. – $EA = BA + INDEX * STEP + DEVIATION$:

ADD AX, [EDX][EDI*4+10h]; Сложить AX с ячейкой памяти.

СТЕКОВАЯ АДРЕСАЦИЯ (можно рассматривать как вариант регистровой косвенной адресации) – в указателе стека ESP (SP) формируется 32-х битовое (16-ти битовое) внутрисегментное смещение для операнда в стековом сегменте :

PUSH	ECX	; Включить в стек содержимое регистра ECX
PUSHFD		; Включить в стек содержимое EFLAGS

PUSH	4000h	; Включить в стек константу
POP	EDX	; Извлечь из стека в регистр
POPCD		; Извлечь из стека в регистр EFLAGS
POP	[ESI]	; Извлечь из стека в ячейку памяти

В таблице 1 показана разница в использовании базовых и индексных регистров для 16-ти и 32-х битовых адресов.

Для обеспечения совместимости ПО процессоров необходимо программы (с 16-ти битовыми командами МП 86 и 286) выполнять на МП 386+ в реальном или защищенном режимах. Процессор определяет размерность адреса, анализируя бит **D** (Default) в дескрипторе сегмента. Если $D=0$, то все длины операндов и эффективных адресов составляют 16 бит. Если $D=1$, – 32 бита. В реальном режиме – 16 бит.

Изменение размерности адреса и данных, задаваемых битом **D**, обеспечивают два префикса, выбираемые перед командами:

- ПРЕФИКС РАЗМЕРНОСТИ ОПЕРАНДА (OperandSize),
- ПРЕФИКС ДЛИНЫ АДРЕСА (AddressSize).

Наличие префикса коммутирует (переключает) размер операнда или размер эффективного адреса на значение, противоположное принимаемому по умолчанию (по биту **D**).

Префиксы могут использоваться совместно с любой инструкцией и в любом режиме – реальном, виртуальном и V86. Префикс длины адреса не обеспечивает размерность адреса более 64 Кбайт в режиме реальной адресации. Адрес свыше 0FFFFh будет рассматриваться как ошибка.

Таблица 1 – Базовые и индексные регистры для 16-ти и 32-х битовых адресов

	16-ти битовый адрес	32-х битовый адрес
Базовый регистр	BX, BP	Любой 32-х битовый РОН
Индексный регистр	SI, DI	Любой 32-х битовый РОН, исключая ESP
Шаг индексации «f»	нет	1, 2, 4, 8
Смещение	0, 8, 16 бит	0, 8, 32 бит

3.2 ИСПОЛЬЗОВАНИЕ СЕГМЕНТНЫХ РЕГИСТРОВ

Основная структура в организации памяти – СЕГМЕНТ.

СЕГМЕНТЫ – блоки памяти переменной длины (от 1 байта до 4 Гбайт), имеющие определенные атрибуты. Три основных типа сегментов – СТЕК, КОМАНДЫ, ДАННЫЕ.

Для компактного кодирования команд и повышения производительности МП – команды не содержат явного указания на используемый

сегментный регистр. Определение сегментного регистра (по умолчанию) производится автоматически в соответствии с табл. 2. Сегментные регистры **FS** и **GS** – не выбираются по умолчанию ни в одной команде и могут быть выбраны только префиксом замены сегмента.

Таблица 2 – Выбор сегментных регистров и внутрисегментного смещения

Тип обращения к памяти	Сегментный регистр	Смещение
Выборка команды	CS	EIP (IP)
Обращение к стеку	SS	ESP (SP)
Адресация операнда	DS (CS,SS,ES,FS,GS)	EA
Элемент цепочки-источника	DS (CS,SS,ES,FS,GS)	ESI (SI)
Элемент цепочки-приемника	ES	EDI (DI)
Операнд с использованием в качестве базового регистра EBP (BP) или ESP (SP)	DS (CS,SS,ES,FS,GS)	EA

Обычно название сегментного регистра указывает на тип информации, для адресации которой он используется. Применение префикса переадресации позволяет явно определять используемый сегментный регистр (см. название регистров в скобках во второй колонке табл. 2), в том числе **FS** и **GS**.

4 КОМАНДЫ ПРОЦЕССОРОВ 386+

КОМАНДЫ ПЕРЕДАЧИ ДАННЫХ

Команды этой группы предназначены для пересылок байт (обозначается B), слов (W) или двойных слов (D) из памяти в регистр, из регистра в память и из регистра в регистр. В одной команде невозможно использование двух операндов, расположенных в памяти (за исключением цепочечных команд и операций со стеком).

Команда **MOV** передает байт, слово или двойное слово из источника в приемник. В поле операндов приемник находится на первом месте, источник – на втором.

Команда **XCHG** осуществляет обмен байт, слов или двойных слов. Различий между приемником и источником нет.

Команда **XLAT** заменяет значение в регистре AL на байт из таблицы, адресуемой регистром (E)BX, причем индексом таблицы служит содержимое регистра AL. Эта команда удобна для преобразования из одного кода в другой.

Команда **LEA** обеспечивает вычисление эффективного адреса EA ячейки памяти в соответствии с указанным способом адресации и загрузку EA (а не содержимого адресуемой ячейки памяти!) в указанный общий регистр.

Команды **LDS, LES...** загружают четыре (или шесть) смежных байта из памяти в адресуемый регистр (16 или 32 бита) и в соответствующий сегментный регистр (16 бит). Слово (двойное слово) операнда источника из ячейки памяти, адресуемой в соответствии с указанным методом адресации, передается в выбранный регистр, а следующее слово – в регистр DS (команда LDS), в регистр ES (команда LES) и т.д.

В таблицах приняты следующие обозначения:

- src – операнд-источник;
- dest – операнд-назначение (операнд-приемник);
- reg – 8/16/32-х битовый регистр;
- reg16/32 – 16/32-х битовый регистр;
- reg16 – только 16-ти битовый регистр;
- reg32 – только 32-х битовый регистр;
- mem – 8/16/32-х битовая ячейка памяти, адресуемая регистрами процессора;
- r/m – 8/16/32-х битовый регистр или ячейка памяти, адресуемая регистрами процессора;
- r/m/i – 8/16/32-х битовый регистр, ячейка памяти, адресуемая регистрами процессора или непосредственный операнд;
- addr – 16/32-х битовый адрес;
- immed – непосредственный операнд.

Таблица 3 – Команды пересылки данных

MOV dest, src	Пересылка (копирование) данных из регистра, памяти или непосредственного операнда в регистр или память
XCHG r/m, reg	Обмен данными (взаимный) между регистрами или регистром и памятью
BSWAP reg32	Перестановка байтов в регистре из порядка младший-старший в порядок старший-младший (486+)
MOVSXB reg, r/m	Копирование байта с расширением до слова или двойного слова, заполняя старшие биты знаком (386+)
MOVSXW reg, r/m	Копирование слова с расширением до двойного слова, заполняя старшие биты знаком (386+)
MOVZXB reg, r/m	Копирование байта с расширением до слова или двойного слова, заполняя старшие биты нулем (386+)
MOVZXB reg, r/m	Копирование слова с расширением до двойного слова, заполняя старшие биты нулем (386+)
XLAT	Трансляция (перекодирование) содержимого AL в значение из таблицы трансляции, адресуемой в (E)BX: AL ← [(E)BX+AL]
LEA reg16/32, mem	Загрузка эффективного адреса в регистр
LDS reg16/32, mem	Загрузка в регистр (двойного) слова из памяти, а в

	DS – следующего 16-ти битового слова
LES reg16/32, mem	Загрузка в регистр (двойного) слова из памяти, а в ES – следующего 16-ти битового слова
LFS reg16/32, mem	Загрузка в регистр (двойного) слова из памяти, а в FS – следующего 16-ти битового слова
LGS reg16/32, mem	Загрузка в регистр (двойного) слова из памяти, а в GS – следующего 16-ти битового слова
LSS reg16/32, mem	Загрузка в регистр (двойного) слова из памяти, а в SS – следующего 16-ти битового слова
IN AL(AX), port8	Ввод в AL (или AX,EAX) из порта с адресом port8
IN AL(AX), DX	Ввод в AL (или AX,EAX) из порта с адресом, хранящимся в DX
OUT port8, AL(AX)	Вывод из AL (или AX,EAX) в порт с адресом port8
OUT DX, AL(AX)	Вывод из AL (или AX,EAX) в порт с адресом, хранящимся в DX

Таблица 4 – Команды работы со стеком

PUSH r/m	Помещение (двойного) слова из регистра или памяти в стек
PUSH immed	Помещение непосредственного операнда в стек (286+)
PUSHA (D)	Помещение в стек регистров AX,CX,DX,BX,SP,BP,SI,DI (286+) или их 32-х битовых расширений (386+)
POP r/m	Извлечение (двойного) слова данных из стека в регистр или память
POPA (D)	Извлечение данных из стека в регистры DI,SI, BP,SP,BX,DX,CX,AX (286+) или их 32-х битовых расширений (386+)
PUSHF (D)	Помещение в стек регистра флагов FLAGS (EFLAGS)
POPF (D)	Извлечение данных из стека в регистр флагов FLAGS (EFLAGS)

Команда **PUSH** передает слово (или двойное слово) из источника в стек, а команда **POP** осуществляет противоположное действие – передает (двойное) слово из стека в приемник. Стек – это область памяти, в которой размещается текущий сегмент стека. Регистр (E)SP содержит смещение последнего включенного в стек слова; оно (смещение) называется **ВЕРШИНОЙ СТЕКА**. По мере включения в стек новых слов они располагаются по меньшим адресам памяти; говорят, что стек растет в направлении уменьшения адресов.

Команда **PUSH** начинается с уменьшения (декремента) содержимого регистра (E)SP на 2 (или 4), т.е. адресует следующее свободное слово (или двойное слово) в стеке; после чего передается (двойное) слово из источника.

Команда **POP** передает слово (или двойное слово) из стека в приемник и завершается увеличением (инкрементом) содержимого (E)SP на 2 (или на 4).

Команда **PUSHA (D)** включает в стек регистры в таком порядке: (E)AX, (E)CX, (E)DX, (E)BX, (E)SP, (E)BP, (E)SI, (E)DI. Включаемым значением регистра (E)SP является то его значение, которое было в нем до выполнения команды PUSHA (D). При выполнении команды PUSHA (D) происходит декремент содержимого регистра (E)SP на 2 (или на 4) при включении в стек каждого регистра.

Извлечение из стека, реализуемое командой **POPA (D)**, вызовет инкремент содержимого регистра (E)SP на ту же величину, поэтому команде POPA (D) не требуется запомненное в стеке содержимое регистра (E)SP.

Таблица 5 – Команды целочисленной арифметики

ADD r/m, r/m/i	Сложение двух операндов: $r/m \leftarrow (r/m + r/m/i)$
XADD r/m, reg	Обмен и сложение (486+)
ADC r/m, r/m/i	Сложение двух операндов с учетом переноса от предыдущей операции: $r/m \leftarrow (r/m + r/m/i + CF)$
INC r/m	Увеличение на 1: $r/m \leftarrow (r/m + 1)$
SUB r/m, r/m/i	Вычитание: $r/m \leftarrow (r/m - r/m/i)$
SBB r/m, r/m/i	Вычитание с заемом: $r/m \leftarrow (r/m - r/m/i - CF)$
DEC r/m	Уменьшение на 1: $r/m \leftarrow (r/m - 1)$
CMP r/m, r/m/i	Сравнение – вычитание без сохранения результата (только установка флагов)
CMPXCHG r/m, reg	Сравнение и обмен данными (486+)
CMPXCHG8B	Сравнение и обмен 8 байт (PENTIUM+)
NEG r/m	Изменение знака операнда (преобразование в дополнительном коде): $r/m \leftarrow (0 - r/m)$
MUL r/m	Умножение AL/AX/EAX на беззнаковое целое значение из r/m
IMUL r/m	Умножение AL/AX/EAX на целое знаковое значение из r/m
IMUL reg16/32, r/m	Знаковое умножение reg16/32 на r/m (помещение результата без расширения разрядности в reg16/32) (16 бит – 286+; 32 бита – 386+)
IMUL reg16/32, r/m, immed	Знаковое умножение r/m на 16/32-х битовый непосредственный операнд и помещение результата без расширения разрядности в reg16/32 (16 бит – 286+; 32 бита – 386+)
DIV r/m	Деление расширенного аккумулятора на беззнаковое число из r/m
IDIV r/m	Знаковое деление расширенного аккумулятора на знаковое целое из r/m
CBW	Знаковое расширение байта в аккумуляторе (AL) до слова: AH ← заполняется битом AL[7]
CWD	Преобразование слова в двойное слово (расширение знака AX в DX) DX ← заполняется битом AX[15]

CWDE	EAX [16...31] ← заполняется битом AX [15]
CDQ	Преобразование двойного слова в – счетверенное: EDX ← заполняется битом EAX [31]
DAA	Коррекция AL после BCD-сложения
DAS	Коррекция AL после BCD-вычитания
AAA	Коррекция AL после ASCII-сложения
AAS	Коррекция AL после ASCII-вычитания
AAM	Коррекция AL после ASCII-умножения
AAD	Коррекция AL, AH перед ASCII-делением

Различие между знаковыми и беззнаковыми числами при выполнении арифметических операций заключается в интерпретации двоичных наборов. Беззнаковые числа – это обычные двоичные числа (все биты значащие), а знаковые числа представлены в дополнительном коде.

Операции сложения и вычитания одинаковы для обоих типов чисел. Единственное отличие заключается в механизме обнаружения выхода за диапазон. Команды сложения и вычитания устанавливают флаг CF, если результат, интерпретируемый как беззнаковое число, оказывается вне диапазона; они же устанавливают флаг OF, если результат, интерпретируемый как знаковое число, выходит за диапазон.

Команда **XADD** – обмена и сложения – обменивает операнды и складывает их. Поэтому на месте операнда-источника остается операнд-получатель, а на месте операнда-получателя формируется сумма.

Команда **NEG** изменяет знак операнда в дополнительном коде.

Команда **CMR** (сравнение) аналогична команде вычитания, но результат нигде не запоминается. Эта команда выставляет флаги, по которым можно определить отношение между двумя операндами: равенство, больше или меньше (см. табл. 6). После команды **CMR** обычно используется команда условного перехода.

Команда **CMRCHG** – сравнение и обмена – воспринимает 3 операнда: операнд-источник в регистре, операнд-получатель в памяти и аккумулятор AL/AX/EAX. Если значения в операнде-получателе и аккумуляторе равны, операнд-получатель заменяется операндом-источником. В противном случае исходное значение операнда-получателя загружается в аккумулятор. Флаги отражают результат, полученный при вычитании операнда-получателя из аккумулятора.

Таблица 6 – Состояние флагов после команды сравнения

Отношение	Знаковые числа	Беззнаковые числа
(dest) > (src)	(ZF=0) & (SF=OF)	(CF=0) & (ZF=0)
(dest) => (src)	SF = OF	CF = 0
(dest) = (src)	ZF = 1	ZF = 1
(dest) <= (src)	(ZF=1) & (SF<>OF)	(CF=1) & (ZF=1)
(dest) < (src)	SF <> OF	CF = 1

Команды умножения могут иметь: одно-, двух- или трехадресную форму.

В одноадресных командах **MUL** и **IMUL** один из сомножителей по умолчанию размещается в аккумуляторе (см. табл. 7), а второй сомножитель указан в команде. Результат умножения в два раза длиннее операндов.

Таблица 7 – Размещение первого множителя и результата умножения

Разрядность операндов	Множитель	Результат	
		Старшая часть	Младшая часть
8	AL	AH	AL
16	AX	DX	AX
32	EAX	EDX	EAX

При двухадресной форме (**IMUL reg16/32,r/m**) или трехадресной форме (**IMUL reg16/32, r/m, immed**) команд умножения со знаком – результат размещается в регистре-приемнике. В этом случае старшие 16 (или 32) разряда произведения при умножении 16-ти (или 32-х) разрядных операндов теряются. Такие команды удобно применять для вычисления адресов элементов массивов.

Команды деления **DIV** и **IDIV** имеют только одноадресную форму, причем разрядность делимого (см. табл. 8) должна вдвое превышать разрядность делителя, указанного в команде.

Знак остатка при выполнении команды **IDIV** устанавливается равным знаку делимого.

Таблица 8 – Размещение делимого и результатов деления

Разрядность делителя	Делимое		Частное	Остаток
	Старшие разряды	Младшие разряды		
8	AH	AL	AL	AH
16	DX	AX	AX	DX
32	EDX	EAX	EAX	EDX

Для подготовки операнда-делимого двойной длины используются команды расширения аккумулятора знаковыми битами. При выполнении команд – **CBW / CWDE** (преобразование байта в слово / преобразование слова в двойное слово с расширением в аккумуляторе) – расширенный операнд остается в аккумуляторе. Команды – **CWD / CDQ** (преобразование слова в двойное слово / преобразование двойного слова в четверенное слово) – расширяют аккумулятор AX или EAX в регистры DX или EDX соответственно, куда заносится старшая половина (расширенный знак) операнда.

Система команд процессоров x86 позволяет выполнять арифметические действия над числами, представленными в ДВОИЧНО-ДЕСЯТИЧНОМ УПАКОВАННОМ ФОРМАТЕ (**BCD код**) или в коде ASCII, используемом при обмене информацией и при вводе с клавиатуры. Для этих чисел допустимы значения от 0 до 9 в младшей тетраде.

Команда **DAA** – ДЕСЯТИЧНОЙ КОРРЕКЦИИ АККУМУЛЯТОРА ПОСЛЕ СЛОЖЕНИЯ BCD-чисел выполняет действия над содержимым AL следующим образом:

- если содержимое младшей тетрады AL больше 9 или установлен флаг AF = 1, то к содержимому AL добавляется 6 ;
- если после этого содержимое старшей тетрады AL стало больше 9 или установлен флаг CF, то число 6 добавляется к старшей тетраде AL.

Аналогичным образом выполняются действия над содержимым AL командой **DAS** – ДЕСЯТИЧНАЯ КОРРЕКЦИЯ ПОСЛЕ ВЫЧИТАНИЯ BCD-чисел:

- если младшая тетрада больше 9 или установлен флаг AF = 1, то из AL вычитается число 6;
- если после этого старшая тетрада больше 9 или установлен флаг CF = 1, то число 6 вычитается из старшей тетрады AL.

Перед выполнением арифметических команд над числами в коде ASCII необходимо очистить старшие тетрады этих чисел. Такие числа называются: распакованными (неупакованными).

Команда **AAA** выполняет коррекцию числа в регистре AL, полученного в результате сложения двух распакованных десятичных операндов. Если содержимое младшей тетрады AL больше 9 или установлен флаг AF = 1, то к содержимому AL добавляется 6; после этого к AH прибавляется 1, очищается старшая тетрада AL, и устанавливаются флаги CF и AF.

Команда **AAS** выполняет коррекцию числа в регистре AL, полученного в результате вычитания двух распакованных десятичных операндов. Если содержимое младшей тетрады AL больше 9 или установлен флаг AF = 1, то из AL вычитается число 6; после этого из AH вычитается 1, очищается старшая тетрада AL, и устанавливаются флаги CF и AF.

Команда **AAM** выполняет коррекцию числа в регистре AL, полученного после умножения двух распакованных десятичных операндов. Содержимое AL делится на 10; частное пересылается в AH, а остаток – в AL.

Команда **AAD** производит коррекцию делимого ДО ВЫПОЛНЕНИЯ команды деления. Для этого содержимое регистра AH умножается на 10 и результат прибавляется к содержимому в AL, старший байт аккумулятора AH очищается. Полученный операнд используется для обычного деления на распакованный делитель.

Логические двухоперандные команды служат для реализации трех булевых функций (результат помещается на место первого операнда):

- AND – поразрядное логическое И;
- OR – поразрядное логическое ИЛИ;
- XOR – поразрядное логическое ИСКЛЮЧАЮЩЕЕ ИЛИ (сумма по модулю 2).

Сюда также относится команда TEST (проверка), которая выполняет поразрядное логическое И, но результат никуда не заносит, а только устанавливаются флаги для выполнения условных переходов.

Команды XOR и SUB позволяют обнулить все биты регистра (регистр должен быть и источником и приемником).

Таблица 9 – Команды логических операций

AND r/m, r/m/i	Побитовое логическое И
TEST r/m, r/m/i	Проверка бит (логическое И без записи результата – установка флагов)
OR r/m, r/m/i	Побитовое логическое ИЛИ
XOR r/m, r/m/i	Побитовое логическое ИСКЛЮЧАЮЩЕЕ ИЛИ
NOT r/m	Побитовая инверсия

Команды сдвигов и циклических сдвигов (табл. 10) выполняют сдвиг 8/16/32-х битового операнда на 1 бит или на произвольное число бит (но не больше длины операнда). Для сдвигов более, чем на один бит, число сдвигов может быть записано предварительно в регистр CL или задано непосредственным операндом в команде (286+). Во всех командах сдвигов последний выдвигаемый бит помещается во флаг CF.



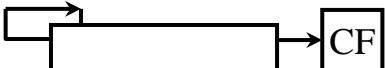
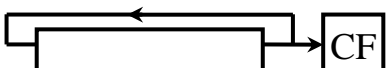
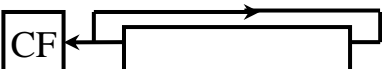
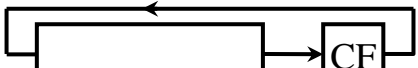
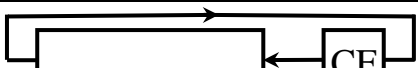
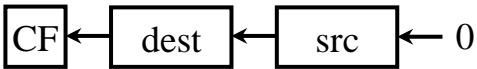
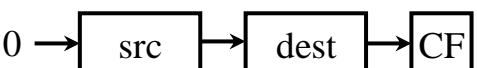
В командах двойного сдвига операндом-приемником (dest) может быть содержимое reg16/32 или mem16/32, операндом-источником (src) – только содержимое РОНа (с разрядностью 16/32). Для сдвигов более, чем на один бит, число сдвигов может быть записано предварительно в регистр CL или задано непосредственным операндом в команде.

Внутри процессора операнды dest и src объединяются в промежуточном регистре двойной длины, содержимое которого логически сдвигается влево или вправо. После сдвига в операнд-приемник (dest) помещаются соответствующие сдвинутые биты промежуточного регистра. Содержимое операнда-источника (src) не изменяется. Можно сказать, что в этих командах сдвигается операнд-приемник (dest) и в его освобождающиеся биты «вдвигается» содержимое операнда-источника (src).

КОМАНДЫ БИТОВЫХ ОПЕРАЦИЙ – отсутствуют в МП 86/286.

Команда – **BT r/m,im8** – или – **BT r/m,reg** – (тестирование бита) выбирает из адресуемого регистра или памяти (r/m) значение определенного бита и копирует его во флаг CF. Номер бита (индекс) определяется значением байта непосредственного операнда или задается содержимым регистра (reg).

Таблица 10 – Команды сдвигов

Команда	Мнемоника	Выполнение команды
Логический сдвиг влево Арифметический сдвиг влево	SHL SAL	
Логический сдвиг вправо	SHR	
Арифметический сдвиг вправо	SAR	
Циклический сдвиг вправо	ROR	
Циклический сдвиг влево	ROL	
Циклический сдвиг вправо через флаг CF	RCR	
Циклический сдвиг влево через флаг CF	RCL	
Двойной сдвиг влево (386+)	SHLD	
Двойной сдвиг вправо (386+)	SHRD	

Когда номер бита (индекс) определен как константа (immed), его диапазон составляет от 0 до 31. Если поле r/m определяет ячейку памяти (размером слово или двойное слово), а номер бита задан содержимым регистра reg, то этот номер бита (индекс) считается целым знаковым числом в диапазоне от -32К до +(32К-1) для 16-ти битовой операции или от -2Г до +(2Г-1) для 32-х битовой операции.

Таблица 11 – Команды битовых операций (386+)

BT r/m, im8 BT r/m, reg	Тестирование бита – загрузка в CF бита с номером (индексом) im8 из r/m. Загрузка в CF бита из r/m с номером из «reg»
BTC r/m, im8 BTC r/m, reg	Тестирование (загрузка в CF) и инверсия бита
BTR r/m, im8 BTR r/m, reg	Тестирование (загрузка в CF) и сброс бита
BTS r/m, im8 BTS r/m, reg	Тестирование (загрузка в CF) и установка в 1 бита
BSF(BSR) reg, r/m	Сканирование бит вперед (назад) в ячейке r/m. В reg загружается индекс первого единичного бита в ячейке r/m.

Аналогичная команда – **BTS** – после копирования устанавливает адресуемый бит в 1. Команда – **BTR** – после копирования сбрасывает бит, а команда – **BTC** – инвертирует.

Команды – **BSF** и **BSR** – производят сканирование содержимого регистра или ячейки памяти (r/m) и заносят в регистр-приемник (reg) номер первого встреченного единичного бита. При выполнении команды – **BSF** – сканирование начинается с младшего разряда, а в команде – **BSR** – со старшего разряда. Если операнд равен нулю (единичные биты отсутствуют), то устанавливается флаг $ZF = 1$. При этом содержимое регистра-приемника будет неопределенным. Если единичный бит найден, то флаг $ZF = 0$.

КОМАНДЫ ОБРАБОТКИ ЦЕПОЧЕК

Под **ЦЕПОЧКОЙ (строкой)** понимается последовательность байт, слов или двойных слов в памяти, а **ЦЕПОЧЕЧНОЙ (строковой) ОПЕРАЦИЕЙ** называется операция, которая выполняется над каждым элементом цепочки. Например, цепочечная передача производит пересылку целой цепочки из одной области памяти в другую. Сокращение времени выполнения цепочечных команд достигается за счет мощного набора примитивных команд, выполняющих ускоренную обработку каждого элемента цепочки и необходимые служебные действия (табл. 12).

Перед выполнением цепочечных команд необходимо:

- загрузить начальный (конечный) адрес цепочки-источника в регистры DS:(E)SI (допускается замена сегмента) (имеются соответствующие команды: LDS и др.);
- загрузить начальный (конечный) адрес цепочки-приемника в регистры ES:(E)DI (командой LES);
- сбросить флаг $DF=0$ (командой CLD), если цепочки обрабатываются по возрастанию адресов, или установить флаг $DF=1$ (командой STD), если цепочки обрабатываются по убыванию адресов;
- при использовании префикса повторения REP в регистр (E)CX загрузить количество повторений цепочечной операции;
- при работе с портами в регистр DX загрузить адрес порта.

Таблица 12 – Примитивы цепочечных (строковых) команд

MOVS	$\text{mem}(\text{DI}) \leftarrow \text{mem}(\text{SI}),$	Модифицировать SI, DI
CMPS	$\text{mem}(\text{SI}) - \text{mem}(\text{DI}), \text{FLAGS},$	Модифицировать SI, DI
SCAS	$A - \text{mem}(\text{DI}), \text{FLAGS},$	Модифицировать DI
LODS	$A \leftarrow \text{mem}(\text{SI}),$	Модифицировать SI
STOS	$\text{mem}(\text{DI}) \leftarrow A,$	Модифицировать DI
INS	$\text{mem}(\text{DI}) \leftarrow \text{port}(\text{DX}),$	Модифицировать DI (286+)
OUTS	$\text{port}(\text{DX}) \leftarrow \text{mem}(\text{SI}),$	Модифицировать SI (286+)

Цепочечный примитив **MOVSB (MOVSW, MOVSD)** – передать элемент цепочки – пересылает байт (слово или двойное слово) из ячейки памяти, смещение которой находится в регистре (E)SI (подразумевается, что цепочка-источник по умолчанию находится в текущем сегменте данных, определяемом регистром DS, но допускается замена сегмента), в ячейку памяти со смещением из (E)DI (цепочка-получатель должна находиться только в сегменте, определяемом регистром ES).

При выполнении цепочечной команды содержимое регистров (E)SI и (E)DI автоматически модифицируется так, чтобы адресовать следующие элементы цепочек. Флаг DF определяет автоинкремент (DF = 0) или автодекремент (DF = 1) индексных регистров. Величина инкремента/декремента зависит от размера элементов и составляет 1, 2 или 4, когда элементами цепочек являются, соответственно, байты, слова или двойные слова.

Если в цепочечную команду добавить префикс повторения: **REP MOVSB**, то примитив MOVSB, будет повторяться с уменьшением (E)CX на 1 (после выполнения примитива) до обнуления (E)CX.

Команда сравнения цепочек **CMPSB (CMPSW, CMPSD)** – производит вычитание байта (слова или двойного слова) цепочки приемника (dest) из соответствующего элемента цепочки-источника (src). В зависимости от результата вычитания устанавливаются флаги (в регистре (E)FLAGS), но сами операнды не изменяются. Индексные регистры-указатели продвигаются на следующие элементы цепочек.

Когда перед командой **CMPS** указан префикс повторения **REPE** или **REPZ**), операция интерпретируется как: «сравнивать, пока не достигнут конец цепочек или пока не найден равный элемент».

При наличии префикса **REPNE** (или **REPNZ**) операция приобретает смысл: «сравнивать, пока не достигнут конец цепочек или пока элементы остаются равными».

Команда сканирования цепочек **SCASB (SCASW, SCASD)** – производит вычитание элемента цепочки (байт, слово или двойное слово) из содержимого аккумулятора AL/AX/EAX. В зависимости от результатов вычитания устанавливаются флаги, но значения операндов не изменяется.

С префиксом **REPE** (или **REPZ**) команду **SCAS** можно использовать для поиска элемента цепочки со значением, отличающимся от заданного в аккумуляторе значения. Префикс **REPNE** (или **REPNZ**) позволяет найти элемент цепочки, значение которого равно значению в аккумуляторе.

Команда **LODSB (LODSW, LODSD)** загружает в аккумулятор (AL/AX/EAX) элемент из цепочки (байт, слово или двойное слово) и продвигает указатель (E)SI на следующий элемент. Обычно эта команда с префиксом повторения не используется.

Команда сохранения аккумулятора в цепочке **STOSB (STOSW, STOSD)** – передает байт (слово или двойное слово) из аккумулятора AL/AX/EAX в элемент цепочки и продвигает регистр-указатель (E)DI на

следующий элемент. С префиксом повторения **REP** эта команда удобна для инициализации цепочки на фиксированное значение.

Команды ввода и вывода цепочек **INSB (INSW, INSD)** и **OUTSB (OUNSW, OUNSD)** как и обычные команды ввода/вывода являются привилегированными.

Команда **INS** вводит данные из порта, адресуемого регистром DX, в ячейку памяти с адресом ES:(E)DI. После ввода операнда производится модификация регистра (E)DI на 1, 2 или 4 с учетом состояния флага направления DF.

Команда **OUTS** выводит данные из ячейки памяти с адресом DS:(E)SI в выходной порт, адрес которого находится в регистре DX. После вывода операнда производится коррекция указателя (E)SI.

Обе эти команды могут использоваться с префиксом повторения **REP**. В этом случае ввод или вывод данных повторяется до обнуления регистра-счетчика (E)CX.

Необходимо отметить, что пять мнемоник префикса повторения **REP, REPE/REPZ, REPNE/REPNZ** определяют только два объектных (машинных) кода префикса (0F2h и 0F3h), а пять мнемоник введены для лучшей передачи содержательного смысла задачи.

КОМАНДЫ РАБОТЫ С ФЛАГАМИ (ФЛАЖКОВЫЕ КОМАНДЫ)

Однobaйтовые команды этой группы позволяют модифицировать некоторые флаги регистра (E)FLAGS (см. табл. 2.19). Остальные флаги могут быть модифицированы после записи содержимого флагового регистра в регистр или ячейку памяти (например, командой PUSHF(D)), с последующим возвратом во флаговый регистр.

Команды, модифицирующие флаг IF, являются IOPL-чувствительными, т.е. выполняющая их программа должна иметь текущий уровень привилегий CPL, меньший или равный содержимому поля IOPL в регистре (E)FLAGS. Если это условие не выполняется, возникает нарушение общей защиты.

Таблица 13 – Команды работы с флагами

CLC	CF ← 0	Сброс флага переноса
CMC	CF ← 1 – CF	Инверсия флага переноса
STC	CF ← 1	Установка флага переноса
CLD	DF ← 0	Сброс флага направления цепочек DF
STD	DF ← 1	Установка флага направления DF
CLI	IF ← 0	Запрет маскируемых аппаратных прерываний
STI	IF ← 1	Разрешение маскируемых аппаратных прерываний
CTS (CLTS)	TF ← 0	Сброс флага переключения задач
LAHF	Загрузка младшего байта регистра флагов в AH	
SAHF	Сохранение AH в младшем байте регистра флагов	

КОМАНДЫ ПЕРЕДАЧИ УПРАВЛЕНИЯ

КОМАНДА БЕЗУСЛОВНОГО ПЕРЕХОДА с общей мнемоникой **JMP** имеет 5 форм, различающихся расстоянием до адреса назначения от текущей команды и способом задания назначения (целевого адреса – target).

- В коротком (SHORT) внутрисегментном переходе двухбайтовая команда **JMP rel8** содержит во втором байте смещение в дополнительном коде (максимально возможный переход: назад – 128 или вперед +127 от адреса команды, находящейся после команды **JMP**).
- Команда прямого внутрисегментного перехода (NEAR) аналогична предыдущей, но полное смещение в дополнительном коде содержит 16 (или 32 бита), которое прибавляется к текущему значению (E)IP. Эта форма команды передает управление в любую точку текущего сегмента кода.
- В команде косвенного внутрисегментного перехода **JMP r/m** адрес целевого назначения (target) загружается в (E)IP из регистра или ячейки памяти.
- Команда прямого межсегментного перехода **JMP prt** содержит непосредственный операнд, содержащий: 16-ти битовый селектор, который загружается в регистр CS, и 16-ти (или 32-х) битовое смещение, загружаемое в (E)IP.
- Команда косвенного межсегментного перехода адресует в памяти полный 32-х (или 48-ми) битовый указатель – селектор: смещение. Селектор загружается в регистр CS, а смещение – в регистр (E)IP.

КОМАНДЫ УСЛОВНЫХ ПЕРЕХОДОВ (табл. 14) осуществляют передачу управления в зависимости от результатов предыдущих операций. Все команды условных переходов производят передачу управления только в пределах текущего сегмента кода (т.е. содержимое сегментного регистра CS не изменяется), если заданное в команде условие удовлетворяется. Переход реализуется прибавлением находящегося в команде смещения (в дополнительном коде) к содержимому регистра (E)IP. В процессорах 86/286 8-ми битовое смещение обеспечивает диапазон перехода от – 128 до +127 байт. В процессорах 386+ наряду с таким смещением допускается также полное 16-ти или 32-х битовое смещение в дополнительном коде. Этим обеспечивается переход в любую точку текущего сегмента кода.

В мнемосодах команд условных переходов при сравнении **чисел со знаком** используются буквы:

– **G** (greater) – больше,
– **L** (less) – меньше.

Для **чисел без знака**:

– **A** (above) – над, выше,
– **B** (below) – под, ниже.

Условие равенства:

– **E** (equal) – равно;

Невыполнение некоторого условия: – **N** (not) – не.

Таблица 14 – Команды передачи управления (переходов)

JMP	target	Безусловный переход к целевому адресу target
J(E)CXZ	target	Условный переход, если (E)CX = 0
LOOP	target	Декремент (E)CX и переход, если (E)CX \neq 0
LOOPE	target	Декремент (E)CX и переход, если (E)CX \neq 0 & ZF = 1
(LOOPZ)	target	
LOOPNE	target	Декремент (E)CX и переход, если (E)CX \neq 0 & ZF = 0
(LOOPNZ)	target	
Jccc	target	Команды условного перехода
CALL	target	Вызов процедуры (подпрограммы)
RET (n)		Возврат из процедуры. Необязательный параметр n задает коррекцию значения указателя стека
SETccc	r/m	Условное заполнение байта. Если выполняется условие «ccc», все биты байта dest (регистра или памяти) устанавливаются в 1, иначе – в 0. Условия «ccc» те же, что и в командах условных переходов (386+)

Для некоторых команд условных переходов зарезервированы два или три альтернативных мнемкода (см. табл. 15), подчеркивающих содержательный смысл проверяемого условия.

Таблица 15 – Кодирование условий перехода

Код поля ссс	Мнемоника поля ссс	Состояние флагов	Условие перехода
0000	O	OF=1	Переполнение
0001	NO	OF=0	Не переполнение
0010	B/NAE/C	CF=1	Ниже / не выше или равно
0011	AE/NB/NC	CF=0	Не ниже / выше или равно
0100	E/Z	ZF=1	Равно / нуль
0101	NE/NZ	ZF=0	Не равно / не нуль
0110	BE/NA	CF=1 & ZF=1	Ниже или равно / не выше
0111	NBE/A	CF=0 & ZF=0	Не ниже или равно / выше
1000	S	SF=1	Есть знак (отрицательный)
1001	NS	SF=0	Нет знака (положительный)
1010	P/PE	PF=1	Есть паритет / четный паритет
1011	NP/PO	PF=0	Нет паритета / нечетный паритет
1100	L/NGE	ZF \neq OF	Меньше / не больше или равно
1101	NL/GE	SF=OF	Не меньше / больше или равно
1110	LE/NG	(SF \neq OF) & ZF=1	Меньше или равно / не больше
1111	NLE/G	SF=(OF & ZF)	Не меньше или равно / больше

КОМАНДЫ ВЫЗОВА ПОДПРОГРАММЫ (процедуры) **CALL** передает управление с автоматическим сохранением в стеке адреса возврата (текущего содержимого IP), т.е. адреса команды, находящейся после команды **CALL**. В конце подпрограммы последняя команда **RET** восстанавливает из стека в регистр IP адрес возврата.

Команда **CALL** имеет такие же формы (относительную, прямую и косвенную), как и команда **JMP**; отсутствует только короткая (SHORT) форма. По воздействию на регистры CS и (E)IP команда **CALL** также соответствует команде **JMP**, но дополнительно включает в текущий сегмент стека адрес возврата с соответствующей коррекцией указателя стека (E)SP.

Команда **RET** допускает указание в поле операнда непосредственной константы `immed16`. В таких командах после извлечения из стека адреса возврата константа `immed16` прибавляется к содержимому регистра (E)SP. В результате в стеке пропускаются параметры, переданные подпрограмме.

Команда заполнения байта по условию (**SETccc r8/m8**) (см. табл. 14) предназначена для того, чтобы сохранить зафиксированное флагами условие для дальнейших вычислений. Мнемоника условия «ccc» полностью совпадает с условием переходов (табл. 15).

КОМАНДЫ ПРЕРЫВАНИЯ

Двухбайтовая команда **INT n** (табл. 16) в начале включает в стек содержимое регистра флагов (E)FLAGS и полный адрес возврата, представленный содержимым регистров CS и (E)IP. Кроме этого сбрасывается в нуль флаг разрешения прерываний IF. После этого осуществляется косвенный переход через элемент «n» дескрипторной таблицы прерываний IDT.

Однбайтовый вариант этой команды **INT 3** называется прерыванием контрольной точки.

Команда прерывания **INTO** эквивалентна команде **INT 4**, если установлен флаг переполнения $OF = 1$. Когда же флаг $OF = 0$, команда **INTO** не производит никаких действий.

Команда возврата из прерывания **IRET** извлекает из стека сохраненные в нем адрес возврата и регистр флагов.

Таблица 2.22 – Команды прерывания

INT n	Выполнение программного прерывания
INT 3	Однбайтовая команда прерывания по типу 3
INTO	Выполнение программного прерывания 4, если $OF=1$
IRET	Возврат из прерывания

ПРАКТИЧЕСКИЕ ЗАДАНИЯ ПО ПРОГРАММИРОВАНИЮ CPU

Задание № 1. Вычислить 7 значений функции:

$$Y = (15 * x^2 + 8 * x - 12) / (4 * x + 5) \quad (x - \text{изменяется от } 3 \text{ с шагом } 3).$$

Результат округлить до целого и разместить в памяти.

Для упрощения программы необходимо переписать функцию в виде:

$$Y = ((15 * x + 8) * x - 12) / (4 * x + 5)$$

```
void main ()                // начало программы на языке C++
{
    long  X=3;               // ячейка памяти для аргумента
    long  REZ[7];            // 7 ячеек памяти для результатов

    _asm{                   ; начало ассемблерной вставки

        lea     EBX, REZ    ; загрузка адреса результатов в регистр EBX
        mov     ECX, 7      ; счетчик количества повторений цикла
m1:    mov     EAX, 4        ; EAX = 4
        imul    X           ; EAX = 4 * x
        add     EAX, 5      ; EAX = 4 * x + 5
        mov     EDI, EAX    ; пересылка знаменателя в регистр EDI
        mov     EAX, 15     ; EAX = 15
        imul    X           ; EAX = 15 * x
        add     EAX, 8      ; EAX = 15 * x
        imul    X           ; EAX = (15 * x + 8) * x
        sub     EAX, 12     ; EAX = (15 * x + 8) * x - 12
        cdq                     ; расширение операнда-делимого в EAX-EDX
        div     EDI         ; частное – EAX , остаток – EDX
        shr     EDI, 1      ; деление знаменателя (делителя) на 2
        cmp     EDI, EDX    ; сравнение половины делителя с остатком
        adc     EAX, 0      ; добавление к частному заема от сравнения
        mov     dword ptr[EBX], EAX ; пересылка результата в память
        add     EBX, 4      ; увеличение адреса результатов
        add     X, 3        ; увеличение аргумента
        loop    m1         ; заикливание по счетчику в ECX
    }                       // окончание ассемблерной вставки
}
```

Задание № 2. Определить номер (n) элемента прогрессии :

$$a_n = 8^n - 5 * n, \text{ при котором сумма элементов прогрессии превысит } 10000.$$

```
void main ()                // начало программы на языке C++
{
    long  N=0;              // ячейка памяти для аргумента
```

```

long S=0;           // ячейка для хранения суммы
long P=1;           // ячейка для накопления 8n
_asm {              ; начало ассемблерной вставки

m1:  inc            N            ; увеличение аргумента
     mov            EAX, 8        ; EAX = 8
     mul            P            ; умножение – 8n
     mov            P, EAX        ; пересылка 8n в ячейку памяти P
     add            S, EAX        ; накопление суммы
     mov            EAX, 5        ; EAX = 5
     mul            N            ; EAX = 5 * n
     sub            S, EAX        ; накопление суммы
     cmp            S, 10000      ; сравнение суммы с 10000
     jc             m1           ; переход, если сумма меньше 10000
     }                        // окончание ассемблерной вставки
}

```

Задание № 3. В памяти задан массив из 5-ти элементов. Поместить в регистр EAX максимальный элемент массива, а в регистр EDI его адрес в памяти.

```

void main () {
    long x[5]={23, 56, 84, 15, 74}    // массив в памяти
    _asm {                            ; начало ассемблерной вставки

        lea         EBX, x            ; начальный адрес массива – в EBX
        mov         ECX, 4            ; счетчик повторений
        mov         EAX, dword ptr[EBX] ; первый элемент – в EAX
        mov         EDI, EBX          ; адрес элемента – в EDI
m2:     add         EBX, 4            ; увеличение адреса
        cmp         EAX, dword ptr[EBX] ; сравнение со следующим элемент.
        jc          m1               ; переход, если меньше
        mov         EAX, dword ptr[EBX] ; больший элемент – в EAX
        mov         EDI, EBX          ; адрес элемента – в EDI
m1:     loop        m2               ; зацикливание по счетчику
    }
}

```

Задание № 4. В памяти задан массив из 8-ми элементов. Отсортировать элементы массива по возрастанию. Пример «пузырьковой сортировки»:

```

void main () {

    long x[8]={7, 23, 56, 33, 84, 15, 11, 74};
    _asm {

```

```

        mov     EDX, 7          ; счетчик внешнего цикла - на 1 меньше
                                   ; количества элементов массива
m3:     lea     EBX, x          ; начальный адрес массива
        mov     ECX, EDX       ; счетчик внутреннего цикла
m2:     mov     EAX, dword ptr[EBX] ; элемент массива – в EAX
        add     EBX, 4
        cmp     EAX, dword ptr[EBX] ; сравнение соседних элементов
        jc      m1            ; переход, если меньше
        xchg    dword ptr[EBX], EAX ; } обмен элементов массива
        mov     dword ptr[EBX-4], EAX ; }
m1:     loop    m2             ; окончание внутреннего цикла
        dec     EDX            ; уменьшение счетчика внешнего цикла
        jnz     m3             ; окончание внешнего цикла
    }}

```

2 ЛАБОРАТОРНАЯ РАБОТА N 2

2.1 ЦЕЛЬ РАБОТЫ

- углубить и закрепить знания по архитектуре процессоров 8086+ и навыки по их программированию;
- приобрести практические навыки в составлении, отладке и выполнении программ, написанных на языке ассемблера для процессоров 8086+.

2.2 САМОСТОЯТЕЛЬНАЯ РАБОТА СТУДЕНТОВ

Перед выполнением лабораторной работы студентам необходимо изучить программную модель и систему команд языка ассемблера процессоров x86+ (CPU).

Изучить основные сведения о работе с программной средой Visual C++, функциональные возможности и режимы работы программы-отладчика.

2.3 ОТЛАДКА ПРОГРАММ НА ЯЗЫКЕ АССЕМБЛЕР В ПРОГРАММНОЙ СРЕДЕ VISUAL C++

После запуска программы «Visual C++»:

- в меню «File» выбрать команду «New»,
- в открывшемся окне выбрать закладку «Projects»,
- на этой закладке выбрать «Win32 Console Application»,
- в поле «Project name» записать имя проекта (например: lab_2_1),
- в поле «Location» выбрать папку для записи проекта.

После формирования папки проекта необходимо ввести программу на языке «C++» с ассемблерной вставкой. Для этого:

- в меню «File» выбрать команду «New»,
- в открывшемся окне выбрать закладку «Files»,
- на этой закладке выбрать «C++ Source File»,
- в поле «File name» записать имя файла.

Для компиляции программы – нажать клавишу «F5».

Для пошаговой отладки программы необходимо:

- установить курсор в начале первой строки ассемблерной вставки,
- в меню «Build» выбрать команду «Start Debug» и вариант «Run to Cursor» (или нажать «CTR-F10»).

Каждый шаг отлаживается нажатием на кнопку «F10».

2.4 ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

2.4.1 Исследовать выполнение арифметических операций. Номер варианта выбирается в соответствии с последней цифрой номера зачетной книжки.

Вариант 1. Вычислить 7 значений функции:

$Y = 7500 / (2 * x^2 + 15)$ (x - изменяется от 3 с шагом 5). Результат округлить до целого и разместить в памяти.

Вариант 2. Вычислить 6 значений функции:

$Y = (6 * x^2 + 12) / (5 * x - 8)$ (x - изменяется от 2 с шагом 4). Результат округлить до целого и разместить в памяти.

Вариант 3. Вычислить 8 значений функции:

$Y = 5 * x^2 + 2 * x - 14$ (x - изменяется от 2 с шагом 4). Результат разместить в памяти.

Вариант 4. Вычислить 5 значений суммы прогрессии для элементов:

$a_n = 2 * n^2 + 5$ (для n - от 4 с шагом 1). Результат разместить в памяти.

Вариант 5. Вычислить 6 значений функции:

$Y = (2500 * x - 8) / (3 * x^2 + 20)$ (x - изменяется от 4 с шагом 3). Результат округлить до целого и разместить в памяти.

Вариант 6. Вычислить 6 значений суммы прогрессии :

$a_n = (3^n) / (n + 5)$ (для n - от 1 с шагом 1). Результат округлить до целого и разместить в памяти.

Вариант 7. Вычислить 8 значений функции :

$Y = (8 * x^2 + 12 * x - 7) / (3 * x + 25)$, (x - изменяется от 2 с шагом 3). Результат округлить до целого и разместить в памяти.

Вариант 8. Вычислить 7 значений функции:

$Y = 7 * x^2 + 12 * x - 32$ (x - изменяется от 3 с шагом 4). Результат разместить в памяти.

Вариант 9. Вычислить 6 значений функции:

$Y = (6^x + 12) / (4 * x^2 - 3)$ (x - изменяется от 1 с шагом 1). Результат округлить до целого и разместить в памяти.

Вариант 10. Вычислить 7 значений суммы прогрессии для элементов:

$a_n = 3 * n^2 + 11$ (для n - от 4 с шагом 1). Результат разместить в памяти.

2.4.2 Исследовать выполнение операций сравнения.

Вариант 1. Найти целое значение аргумента, при котором функция

$Y = 20000 / (8 * x^2 + 25)$ станет меньше 20.

Вариант 2. Определить номер (n) элемента прогрессии :

$a_n = n^2 + 6 * n + 28$, при котором сумма элементов прогрессии превысит 1000.

Вариант 3. Найти целое значение аргумента, при котором функция

$Y = 15 * x^2 + 11 * x - 16$ станет больше 2000.

Вариант 4. Найти целое значение аргумента, при котором функция $Y = (7^x) / (5 * x^2)$ превысит 300.

Вариант 5. Найти целое значение аргумента, при котором функция $Y = (2000 + x) / (8 * x^2 + 25)$ станет меньше 10.

Вариант 6. Найти целое значение аргумента, при котором функция $Y = 9 * x^2 - 8 * x + 15$ станет больше 1000.

Вариант 7. Определить номер (n) элемента прогрессии :
 $a_n = 5^n + 8 * n$, при котором сумма элементов прогрессии превысит 20000.

Вариант 8. Найти целое значение аргумента, при котором функция $Y = 7 * x^2 + 25 * x - 27$ станет больше 3000.

Вариант 9. Найти целое значение аргумента, при котором функция $Y = 300 * x / (8^x + 14)$ станет меньше 5.

Вариант 10. Определить номер (n) элемента прогрессии :
 $a_n = 3 * n^2 - 5 * n + 12$, при котором сумма элементов прогрессии превысит 1500.

2.4.3 Исследовать выполнение операций над массивами в памяти.

Вариант 1. В памяти задан массив из 10-ти элементов. Поместить в регистр EAX минимальный элемент массива, а в регистр EDX его адрес в памяти.

Вариант 2. В памяти задан массив из 10-ти элементов. Сохранить в регистре ESI количество отрицательных элементов.

Вариант 3. Рассчитать и сохранить в памяти элементы массива, заданные функцией: $Y = n!$ (для n от 1 до 8)

Вариант 4. В памяти задан массив из 10-ти элементов. Заменить эти числа произведением их старшего и младшего слова.

Вариант 5. В памяти задан массив из 8-ми элементов. Поместить в регистр EAX максимальный элемент массива, а в регистр ESI его адрес в памяти.

Вариант 6. В памяти задан массив из 9-ти элементов. Отсортировать элементы массива по возрастанию.

Вариант 7. В памяти задан массив из 10-ти элементов. Сохранить в регистре ESI количество нечетных элементов.

Вариант 8. В памяти задан массив из 12-ти элементов. Сохранить в регистре EAX среднее арифметическое этих элементов. Результат округлить до целого.

Вариант 9. В памяти задан массив из 10-ти элементов. Сохранить в регистре ESI количество единичных битов во всех элементах.

Вариант 10. В памяти задан массив из 11-ти элементов. Отсортировать элементы массива по убыванию.

2.5 КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Программная модель 32-х разрядных процессоров x86 (386+).
2. Перечислите форматы данных процессоров x86.
3. Зачем нужны форматы двоично-десятичных чисел?
4. Перечислите методы (способы) адресации данных в процессорах x86.
5. Перечислите команды пересылки данных и расположение операндов-приемников и операндов-источников.
6. Перечислите арифметические и логические команды.
7. Где расположены операнды в командах умножения и деления? Куда записываются результаты умножений и делений?
8. Какие команды позволяют обрабатывать десятичные данные без перевода их в двоичный формат?
9. Как выполняются команды сдвигов?
10. Перечислите цепочечные (строковые) команды и особенности их выполнения.
11. Как выполняются команды условных и безусловных переходов?
12. Чем отличаются команды JMP и CALL?

РЕКОМЕНДОВАННАЯ ЛИТЕРАТУРА

1. Гук М., Юров В. Процессоры PENTIUM 4, ATHLON и другие – СПб: Питер, 2001.– 512с.
2. Юров В. Assembler – СПб.: Питер, 2001.– 624с.
3. Григорьев В.Л. Микропроцессор i486. Архитектура и программирование. В 4-х книгах.-М.: Гранал, 1993.