УЧЕБНО-НАУЧНО-ПРОИЗВОДСТВЕННЫЙ КОМПЛЕКС
«МЕЖДУНАРОДНЫЙ УНИВЕРСИТЕТ КЫРГЫЗСТАНА»

«УТВЕРЖДЕНО»
Ректор НОУ УНПК «МУК»
к.т.н., доцент Савченков Е.Ю.
____
«16» 10 2018 г.

БАКАЛАВРИАТ

Кафедра «Компьютерные информационные системы и управление»

Учебно-методический комплекс дисциплины

ЭВМ и периферийные устройства

Направление: **710100 «Информатика и вычислительная техника»**

Профиль: **Компьютерные информационные системы в бизнесе**

Академическая степень **- бакалавр**

Форма обучения **(очная)**

График проведения модулей 7-семестр

неделя	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
лекц. зан.	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
прак./лаб. зан.	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2

«РАССМОТРЕНО»
Протокол заседания кафедры
«КИСиУ»
№ 2 от 16.10.2018
Зав. кафедрой д.т.н., проф. Миркин Е.Л.
____

«СОГЛАСОВАНО»
Проректор по академ. вопросам
проф. Мадалиев М.М.
____

Составитель __________ к.ф.-м..н., и.о.доцент Красниченко Л.С.

Директор Научной библиотеки __________ Асанова Ж.Ш.

БИШКЕК 2018

ОГЛАВЛЕНИЕ

АННОТАЦИЯ

Дисциплина «ЭВМ и периферийные устройства» представляет собой специализированный курс, который является одним из важнейших при подготовке специалистов в области информационных технологий. Дисциплина включает в себя следующие разделы: основные понятия и определения; основы языка ассемблер запоминающие устройства процессора, АЛУ, устройства управления, процессоры и системные средства ЭВМ и оценка характеристик этих устройств, особенности архитектуры ЭВМ различных классов. Данный курс состоит из четырех модулей, итоговый контроль в виде экзамена.

УЧЕБНО-МЕТОДИЧЕСКИЙ КОМПЛЕКС ДИСЦИПЛИНЫ (МОДУЛЕЙ)

1. Пояснительная записка

1.1. Миссия и стратегия

Миссия НОУ УНПК "МУК" – подготовка международно - признанных, свободно мыслящих специалистов, открытых для перемен и способных трансформировать знания в ценности на благо развития общества.

Видение НОУ УНПК «МУК»- создание динамичного и креативного университета с инновационными научно-образовательными программами и с современной инфраструктурой, способствующие достижению академических и профессиональных целей.

Стратегии развития - модернизация образовательной деятельности университета – совершенствование образовательного процесса в соответствии с требованиями Болонского процесса.

1.2. Цель и задачи дисциплины

Цель дисциплины:

предоставление обучаемым знаний по вопросам функциональной и структурной организации ЭВМ, ее составных частей с применением современных информационных технологий; усвоение этих знаний студентами, а также формирование у них мотивации к самообразованию за счет активизации самостоятельной познавательной деятельности.

Задачи дисциплины:

В результате усвоения материала настоящего курса студенты должны знать:

- принципы программной организации процессора;
- функциональную и структурную организацию ЭВМ;
- принципы построения основных устройств ЭВМ;
- важнейшие этапы и тенденции в развитии цифровой, аналоговой и гибридной вычислительной техники;

1.3. Формируемые компетенции, а также перечень планируемых результатов обучения по дисциплине

Дисциплина направлена на формирование следующих компетенций:

В результате освоения дисциплины обучающийся должен демонстрировать следующие результаты образования:

1. Дисциплина направлена на формирование следующих компетенций:

- ✓ способен разрабатывать бизнес-планы и технические задания на оснащение отделов, лабораторий, офисов компьютерным и сетевым оборудованием (ПК-1);
- ✓ способен сопрягать аппаратные и программные средства в составе информационных и автоматизированных систем (ПК-10);
- ✓ способен инсталлировать программное и аппаратное обеспечение для информационных и автоматизированных систем (ПК-11).

В результате освоения дисциплины обучающийся должен демонстрировать следующие результаты образования:

Знать:

- основы программного построения и архитектуры ЭВМ
- особенности организации различных типов ЭВМ;
- функциональную и структурную организацию центрального процессора, памяти компьютера;

- организацию прерываний и ввода-вывода;
- современное состояние и тенденции развития ЭВМ;

Уметь:

- выбирать, комплексировать и тестировать аппаратные средства вычислительных систем:
- проводить анализ всего многообразия типов ЭВМ с целью выбора наиболее приемлемого варианта для конкретного использования;
- проводить сравнительный анализ параметров основных технических средств ЭВМ (процессора, памяти);
- уметь выбирать базовую конфигурацию компьютера;
- использовать сеть Internet для работы с Web-серверами ведущих фирм производителей средств вычислительной техники;

Владеть: навыками конфигурирования компьютеров различного назначения

1.4. Место дисциплины (модулей) в структуре ООП ВПО

Дисциплина «ЭВМ и периферийные устройства» является частью профессионального цикла (блока) дисциплин учебного плана по направлению подготовки 710100 «Информатика и вычислительная техника» подготовки бакалавров (специализации Компьютерные информационные системы).

Для освоения дисциплины (модулей) необходимы компетенции, сформированные в ходе изучения следующих дисциплин и прохождения практик: основные разделы математики, программирования.

В результате освоения дисциплины (модулей) формируются компетенции, необходимые для изучения следующих дисциплин и прохождения практик: написание выпускной квалифицированной работы.

2. Структура дисциплины

Общая трудоемкость дисциплины составляет 4 кредита, 120ч., в том числе аудиторная работа обучающихся с преподавателем 60 ч., самостоятельная работа обучающихся 40ч., 20ч-СРСП.

№ п/п	***Раздел, Темы Дисциплины***	***Виды учебной работы, включая самостоятельную работу студентов и трудоемкость (в часах)***				***Формы текущего контроля успеваемости (по неделям семестра) Форма промежуточной аттестации (по семестрам)***
		Лекции	***Лаб.***	***СРС***	***СРСП***	
1	**Раздел 1. Арифметические основы ЭВМ**.	***3***	***9***	***10***	***5***	

2	Тема 1. Системы счисления.	*2*	*4*	*5*	*2*	
	Тема 2. Представление чисел в ЭВМ.	*2*	*5*	*45*	*2*	*Сдача модуля*
3	**Раздел 2. Основы теории логического проектирования цифровых устройств.**	***4***	***12***	***10***	***5***	
	Тема 1. Конечные автоматы	*2*	*6*	*5*	*2*	
4	Тема 2. Булева алгебра и минимизация функций	*2*	*6*	*5*	*3*	*Сдача модуля*
5	**Раздел 3. Элементы и функциональные узлы ЭВМ. (4 ч)**	***4***	***12***	***10***	***5***	
	Тема 1. Физическое устройство логических элементов ЭВМ. (2 ч)	*2*	*6*	*5*	*2*	
6	Тема 2. Логические схемы основных узлов в ЭВМ. (2 ч)	*2*	***6***	***5***	***3***	*Сдача модуля*
7	**Раздел 4. Процессоры ЭВМ.**	***4***	***12***	***10***	***5***	
8	Тема 1. Процессорные устройства.	*2*	*4*	*3*	*1*	
9	Тема 2. Арифметико-логические устройства	*1*	***4***	***3***	***1***	
10	Тема 3. ЦУУ. Выполнение программ процессором	*1*	*4*	*4*	*3*	*Сдача модуля*
		15	*45*	*40*	*20*	

3. Содержание дисциплины

Раздел 1. Арифметические основы ЭВМ.

Тема 1. Системы счисления.

Тема 2. Представление чисел в ЭВМ.

Системы исчисления чисел. Формы представления чисел в ЭВМ (числа с фиксированной точкой и с плавающей запятой). Прямой, обратный и дополнительный коды чисел. Двоичная и десятичная арифметика. Алгоритмы сложения, вычитания, умножения и деления чисел.

Раздел 2. Основы теории логического проектирования цифровых устройств.

Тема 1. Конечные автоматы

Конечные автоматы комбинационного и последовательного типа. Синхронные и асинхронные элементы. Структурный синтез цифровых автоматов. Основные теоремы булевой алгебры одной и нескольких переменных. Булевы функции и способы их представления.

Тема 2. Булева алгебра и минимизация функций

Способы минимизации функций. Функционально полные системы логических элементов. Основные понятия логического синтеза комбинационных и последовательных устройств. Автоматы Мура и Мили. Кодировочные таблицы входов, переходов и выходов.

Раздел 3. Элементы и функциональные узлы ЭВМ.

Тема 1. Физическое устройство логических элементов ЭВМ

Физические формы представления информации в ЭВМ. Системы цифровых элементов, основные характеристики, классификация. Логические и запоминающие элементы. Условные графические обозначения.

Тема 2. Логические схемы основных узлов в ЭВМ.

Системы логических элементов, принцип действия, состав, основные характеристики. Реализация логических схем на заданной системе элементов.

Триггеры интегральных комплексов элементов, регистры, счетчики, дешифраторы, мультиплексоры, сумматоры. Основные типы, классификация, принципы действия и логическая структура, таблицы переходов. Условные графические обозначения.

Раздел 4. Процессоры ЭВМ.

Тема 1. Процессорные устройства.

Процессорные устройства. Назначение и структура процессора. Характеристики основных блоков процессоров. Работа процессора.

Тема 2. Арифметико-логические устройства

Арифметико-логические устройства. Алгоритмы выполнения логических и арифметических операций с фиксированной точкой в АЛУ, в двоичной и десятичной системах исчисления. Методы выполнения операций умножения. Алгоритмы выполнения операций деления.

Структура оперативной части для выполнения операций над числами с фиксированной точкой. Особенности взаимодействия узлов и блоков АЛУ при выполнении арифметических операций над числами с плавающей запятой.

Тема 3. ЦУУ. Выполнение программ процессором

Центральное устройство управления. Основные функции ЦУУ.

Выполнение программ в процессоре. Понятие операции, такта, микрооперации, микропрограммы. Операционные устройства процессоров для обработки командной информации. Организация порядка выборки команд. Выполнение команд условного и безусловного переходов. Операции над индексами. Выборка операндов при непосредственной и косвенной адресации. Кодирование команд, форматы команд.

Устройства управления процессором (жесткая логика). Способы формирования сигналов управления микрооперациями.

Микропрограммный принцип построения УУ. Структура микропрограммного УУ. Организация чтения микропрограммы из памяти.

Средства мультипрограммной работы процессора. Принципы организации прерываний. Многоуровневые системы прерываний. Приоритеты. Слово состояния программы. Структурные схемы блоков прерываний

4. Конспект лекций – ПРИЛОЖЕНИЕ 1

5. Информационные и образовательные технологии

Информационные и образовательные технологии

№ п/п	*Наименование раздела*	*Виды учебной работы*	*Формируемые компетенции (указывается код компетенции)*	*Информационные и образовательные технологии*
1	***2***	***3***	***4***	***5***
2		Лекция Лабораторная работа Самостоятельная работа	(ПК-1) (ПК-10) (ПК-11)	**Лекция-визуализация** с применением проектора **Лабораторная работа** согласно текущей темы **Подготовка к занятию** с использованием электронного курса лекций
3		Лекция Лабораторная работа Самостоятельная работа	(ПК-1) (ПК-10) (ПК-11)	**Лекция-визуализация** с применением проектора **Лабораторная работа** согласно текущей темы **Подготовка к занятию** с использованием электронного курса лекций
6		Лекция Лабораторная работа Самостоятельная работа	(ПК-1) (ПК-10) (ПК-11)	**Лекция-визуализация** с применением проектора **Лабораторная работа** согласно текущей темы **Подготовка к занятию** с использованием электронного курса лекций
7		Лекция Лабораторная работа Самостоятельная работа	(ПК-1) (ПК-10) (ПК-11)	**Лекция-визуализация** с применением проектора **Лабораторная работа** согласно текущей темы **Подготовка к занятию** с использованием электронного курса лекций
8		Лекция Лабораторная работа Самостоятельная работа	(ПК-1) (ПК-10) (ПК-11)	**Лекция-визуализация** с применением проектора **Лабораторная работа** согласно текущей темы **Подготовка к занятию** с использованием электронного курса лекций

10		Лекция Лабораторная работа Самостоятельная работа	(ПК-1) (ПК-10) (ПК-11)	**Лекция-визуализация** с применением проектора **Лабораторная работа** согласно текущей темы **Подготовка к занятию** с использованием электронного курса лекций
11		Лекция Лабораторная работа Самостоятельная работа	(ПК-1) (ПК-10) (ПК-11)	**Лекция-визуализация** с применением проектора **Лабораторная работа** согласно текущей темы **Подготовка к занятию** с использованием электронного курса лекций
		Лекция Лабораторная работа Самостоятельная работа	(ПК-1) (ПК-10) (ПК-11)	**Лекция-визуализация** с применением проектора **Лабораторная работа** согласно текущей темы **Подготовка к занятию** с использованием электронного курса лекций
		Лекция Лабораторная работа Самостоятельная работа	(ПК-1) (ПК-10) (ПК-11)	**Лекция-визуализация** с применением проектора **Лабораторная работа** согласно текущей темы **Подготовка к занятию** с использованием электронного курса лекций

6. Фонд оценочных средств для текущего, рубежного и итогового контролей по итогам освоению дисциплины (модулей)

6.1. Перечень компетенций с указанием этапов их формирования в процессе освоения дисциплины

Перечень компетенций с указанием этапов их формирования в процессе освоения дисциплины представляется в виде таблицы:

№ п/п	***Контролируемые разделы дисциплины (модулей)***	***Код контролируемой компетенции (компетенций)***	***Наименование оценочного средства***
1	Тема 1. Системы счисления.	((ПК-1) (ПК-10)	Практическое задание

2	Тема 2. Представление чисел в ЭВМ.	(ПК-11)	Коллоквиум (Вопросы по темам дисциплины)
3	Тема 3. Конечные автоматы	(ПК-1) (ПК-10) (ПК-11) (	Практическое задание «*Исследование программной архитектуры ПК»* Коллоквиум (Вопросы по темам дисциплины)
4	Тема 4. Булева алгебра и минимизация функций		
5	Тема 5. Физическое устройство логических элементов ЭВМ		
6	Тема 6. Логические схемы основных узлов в ЭВМ.	((ПК-1) (ПК-10) (ПК-11)	Практическое задание *«Работа с машинными командами и командами ассемблера с помощью отладчика debug»* Коллоквиум (Вопросы по темам дисциплины) Контрольная работа
7	Тема 7. Процессорные устройства.		
8	Тема 8. Арифметико-логические устройства	((ПК-1) (ПК-10) (ПК-11)	Практическое задание *«Работа с машинными командами и командами ассемблера с помощью отладчика debug»* Коллоквиум (Вопросы по темам дисциплины) Тест
9	Тема 9. ЦУУ. Выполнение программ процессором		

6.2. Методические материалы, определяющие процедуры оценивания знаний, умений, навыков и (или) опыта деятельности

Методические материалы составляют систему текущего, рубежного и итогового (экзамена) контролей освоения дисциплины (модулей), закрепляют виды и формы текущего, рубежного и итогового контролей знаний, сроки проведения, а также его сроки и формы проведения (устный экзамен, письменный экзамен и т.п.). В системе контроля указывается процедура оценивания результатов обучения, при использовании балльно-рейтинговой системы приводится таблица с баллами и требованиями к пороговым значениям достижений по видам деятельности обучающихся; показывается механизм получения оценки (из чего складывается оценка по дисциплине (модулю).

Текущий контроль осуществляется в виде опроса, участие в дискуссии на семинаре, выполнение самостоятельной - оценивается до **80 баллов**.

Рубежный контроль (сдача модулей) проводится преподавателем и представляет собой письменный контроль, либо компьютерное тестирование знаний по теоретическому и практическому материалу. Контрольные вопросы рубежного контроля включают полный объём материала части дисциплины (модулей), позволяющий оценить знания, обучающихся по изученному материалу и соответствовать УМК дисциплины, которое оценивается до **20 баллов**.

Итоговый контроль (экзамен) знаний принимается по экзаменационным билетам, включающий теоретические вопросы и практическое задание, и оценивается до **20 баллов**.

Форма контроля	Срок отчетности	Макс. количество баллов	
		За одну работу	Всего
Текущий контроль: - Прием лабораторных работ -опрос - посещаемость	 1,2,3,4недели 1, ,2,3,4недели 1, ,2,3,4неделя	 8 баллов 6 баллов 2 балла	 До 40 баллов До 30 баллов 10 баллов
Рубежный контроль: (сдача модуля)	4 неделя	100%×0,2=20 баллов	
Итого за I модуль			До 100 баллов

Форма контроля	Срок отчетности	Макс. количество баллов	
		За одну работу	Всего
Текущий контроль: - Прием лабораторных работ -опрос - посещаемость	 5,6,7,8 недели 5,6,7,8 недели 5,6,7,8 недели	 10 баллов 6 баллов 2 балла	 До 40 баллов До 30 баллов 10 баллов
Рубежный контроль: (сдача модуля)	**8 неделя**	100%×0,2=20 баллов	
Итого за II модуль			До 100 баллов

Форма контроля	Срок отчетности	Макс. количество баллов	
		За одну работу	Всего
Текущий контроль: - Прием лабораторных работ -опрос - посещаемость	 9, 10, 11 недели 9, 10, 11 недели 9, 10, 11 недели	 8 баллов 6 баллов 2 балла	 До 40 баллов До 30 баллов 10 баллов
Рубежный контроль:	**11 неделя**	100%×0,2=20 баллов	

(сдача модуля)			
Итого за III модуль			До 100 баллов

Форма контроля	Срок отчетности	Макс. количество баллов	
		За одну работу	Всего
Текущий контроль: - Прием лабораторных работ -опрос - посещаемость	 12,13,14 недели 12,13,14 недели 12,13,14 недели	 10 баллов 6 баллов 2 балла	 До 40 баллов До 30 баллов 10 баллов
Рубежный контроль: (сдача модуля)	**14 неделя**	100%×0,2=20 баллов	
Итого за IV модуль			До 100 баллов

Экзаменатор выставляет по результатам балльной системы в семестре экзаменационную оценку без сдачи экзамена, набравшим суммарное количество баллов, достаточное для выставления оценки от 55 и выше баллов – автоматически (при согласии обучающегося).

Полученный совокупный результат (максимум 100 баллов) конвертируется в традиционную шкалу:

Рейтинговая оценка (баллов)	Оценка экзамена
От 0 - до 54	неудовлетворительно
от 55 - до 69 включительно	удовлетворительно
от 70 – до 84 включительно	хорошо
от 85 – до 100	отлично

6.1.Описание показателей и критериев оценивания компетенций на различных этапах их формирования, описание шкал оценивания

Текущий контроль (0 - 80 баллов)

При оценивании посещаемости, опроса и приема лабораторных работ из расчета на одну неделю учитываются:

- посещаемость (2 балла одно занятие (10 баллов за модуль)
- степень раскрытия содержания материала (2.8 балла одно занятие (14 баллов за модуль);
- изложение материала (грамотность речи, точность использования терминологии и символики, логическая последовательность изложения материала (2.8 балла одно занятие (14 баллов за модуль);
- знание теории изученных вопросов (2.8 балла одно занятие (14 баллов за модуль);

- сформированность и устойчивость используемых при ответе умений и навыков (2.8 балла одно занятие (14 баллов за модуль);
-точность решения задачи (2.8 балла одно занятие (14 баллов за модуль).

Рубежный контроль (0 – 20 баллов)

При оценивании контрольной работы учитывается:
- полнота выполненной работы (задание выполнено не полностью и/или допущены две и более ошибки или три и более неточности) – 8 баллов;
- обоснованность содержания и выводов работы (задание выполнено полностью, но обоснование содержания и выводов недостаточны, но рассуждения верны) – 14 баллов;
- работа выполнена полностью, в рассуждениях и обосновании нет пробелов или ошибок, возможна одна неточность - 17 баллов.
- работа выполнена полностью, в рассуждениях и обосновании нет пробелов или ошибок - 20 баллов.

При оценивании теста учитывается:
- полнота выполненной работы (задание выполнено не полностью и/или допущены две и более ошибки или три и более неточности) – до 20 баллов;

Итоговый контроль (экзаменационная сессия) - ИК = Бср × 0,8+Бэкз × 0,2

При проведении итогового контроля обучающийся должен ответить на 3 вопроса (два вопроса теоретического характера и один вопрос практического характера).

При оценивании ответа на вопрос теоретического характера учитывается:
- теоретическое содержание не освоено, знание материала носит фрагментарный характер, наличие грубых ошибок в ответе (2 балла);
- теоретическое содержание освоено частично, допущено не более двух-трех недочетов (5 баллов);
- теоретическое содержание освоено почти полностью, допущено не более одного-двух недочетов, но обучающийся смог бы их исправить самостоятельно (8 баллов);
- теоретическое содержание освоено полностью, ответ построен по собственному плану (10 баллов).

При оценивании ответа на вопрос практического характера учитывается:
- ответ содержит менее 20% правильного решения (3 балла);
- ответ содержит 21-89 % правильного решения (7 баллов);
- ответ содержит 90% и более правильного решения (10 баллов).

6.2.Типовые контрольные задания или иные материалы, необходимые для оценки знаний, умений, навыков и (или) опыта деятельности.

Раздел УМК включает образцы оценочных средств, примерные перечни вопросов и заданий в соответствии со структурой дисциплины и системой контроля.

Контрольные вопросы

1. Понятия информации и сигнала. Виды сигналов
2. Понятие системы обработки данных (СОД). Определение вычислительной системы (ВС).
3. Основные понятия теории систем: функция системы, структура системы, организация системы, элемент системы.

4. Основные факторы, определяющие принципы организации ЭВМ: элементная база, назначение ЭВМ. Неймановская архитектура ЭВМ.
5. Основные технические характеристики ВС: операционные ресурсы, емкость памяти, быстродействие устройств, надежность, стоимость.
6. Режимы работы ЭВМ: однопрограммный, многопрограммный. Достоинства и недостатки.
7. Основные средства мультипрограммирования: управляющие программы ОС, ОП, система прерываний, средства защиты информации.
8. Организация системы прерываний.
9. Представление информации в ЭВМ. Позиционные системы счисления. Правила двоичной арифметики.
10. Формы представления чисел в ЭВМ.
11. Прямой, обратный и дополнительный коды.
12. Организация виртуальной памяти ЭВМ.
13. Структурная организация ЭВМ. Классы устройств ЭВМ: процессор, операционные устройства, ЗУ, устройства ввода/вывода.
14. Типовые структуры однопроцессорных ВС.
15. Структуры мультипроцессорных и мультимашинных вычислительных комплексов.
16. Назначение и структура процессора.
17. Цикл выполнения команд.
18. Конвейерная организация процессора. Особенности организации современных ЦП.
19. Эволюция способов организации процессора. Распараллеливание уровня команд.
20. Понятие VLIW и мультискалярная организация процессоров.
21. Функциональная организация операционных устройств. Принцип микропрограммного управления.
22. Построение кэш-памяти.
23. Схемный контроль. Принципы и реализация
24. Понятие кодового расстояния. Код Хэмминга
25. Вычислительные системы. Их классификация. Характеристики основных типов
26. Способы преодоления недостатков Неймановской архитектуры ПК.
27. Организация оперативной памяти ЭВМ.
28. Структурная организация ЭВМ. Магистрально-модульный принцип построения ЭВМ. Понятие интерфейса.

Тематика рефератов

Тест: Приложение III

Контрольная работа

Самостоятельная работа студентов

7. Учебно-методическое и информационное обеспечение дисциплины

7.1. Список источников и литературы

Основные учебники

1. Жмакин А.П. Архитектура ЭВМ: Учебное пособие / А.П. Жмакин. - СПб.: БХВ - Петербург, 2006 - 320с

2. Олифер В.Г. Компьютерные сети : Принципы, технологии, протоколы / В.Г. Олифер, Н.А. Олифер. - СПб: Питер, 2010 - 944с.
3. Таненбаум Э. Компьютерные сети / Э. Таненбаум. - СПб: Питер, 2010 - 992с.
4. Чекмарев Ю.В. Вычислительные системы, сети и телекоммуникации / Ю.В. Чекмарев.- ДМК Пресс, 2009 – 184с.

Дополнительная литература

1. Вильховченко О.С Современный компьютер: устройство, выбор, модернизация / О.С. Вильховченко. - СПб: Питер, 2000 - 512с.
2. Гук М. Аппартные средства локальных сетей : Энциклопедия / М. Гук. - СПб: Питер, 2004 - 573с.
3. Гук М. Аппаратные средства IBM PC. / М. Гук. - СПб: Питер, 2000 - 816с.
4. Колесниченко О.В. Аппаратные средства PC / О.В. Колесниченко, И.В. Шишигин. - СПб.: БХВ - Петербург, 2004 - 1152с.
5. Косцов А. Все о персональном компьютере: Большая энциклопедия / А. Косцов, В. Косцов. - М.: Мартин, 2005 - 720с.
6. Леонтьев В.П. Новейшая энциклопедия персонального компьютера 2003 / В.П. Леонтьев. - М.: ОЛМА-Пресс, 2003 - 920с
7. Мельников Д.А Информационные процессы в компьютерных сетях : Протоколы, стандарты, интерфейсы, модели / Д.А. Мельников. - М.: Кудиц-Образ, 2001 - 256с.

7.2.Перечень ресурсов информационно-телекоммуникационной сети «Интернет», необходимый для освоения дисциплины (модулей)

1. https://www.intuit.ru/search
2. https://www.twirpx.com/ Библиотека все для студента
3. https://uk.sagepub.com/en-gb/asi/home
4. https://uk.sagepub.com/en-gb/asi/sage-premier
5. https://www.nejm.org/
6. https://uk.sagepub.com/en-gb/asi/imeche
7. http://global.oup.com/?cc=kg
8. https://www.cambridge.org
9. https://www.intellectbooks.co.uk/journals/index/
10. http://iopscience.iop.org/journalList
11. https://royalsociety.org/journals/
12. https://www.elibrary.imf.org/?redirect=true
13. https://www.elgaronline.com/page/70/journals
14. http://www.dukejournals.org/
15. http://www.iprbookshop.ru/
16. http://kyrlibnet.kg/ru/
17. http://biblioteka.kg/

8. Перечень учебно-методического обеспечения для самостоятельной работы обучающихся

8.1.Планы практических (семинарских) и лабораторных занятий. -ПРИЛОЖЕНИЕ II

8.2. Методические указания по организации и проведению лабораторных

занятий

8.3. Методические указания для обучающихся по освоению дисциплины (модулей)

Методические указания предназначены для рационального распределения времени студента по видам **самостоятельной работы** и разделам дисциплины. Они составляются на основе сведений о трудоемкости дисциплины, ее содержании и видах работы по ее изучению, а также учебно-методического и информационного обеспечения. В раздел включаются: рекомендации по изучению дисциплины (модулей) или отдельных тематических разделов, вопросы и задания для самостоятельной работы, материалы, необходимые, для подготовки к занятиям (разделы книг, статьи и т.д.). Раздел может быть представлен в табличной форме.

8.4.Методические рекомендации по подготовке отчетов по лабораторным работам

Требования при оформлении лабораторных работ:

1. Требования
   - Первая страница Титульный лист
   - Условия задачи, цели, этапы выполнения
   - Программный код
   - Графики
   - Результаты
   - Выводы

Правила оформления лабораторных работ:

- текст печатается на странице формата А4;
- шрифт – Times New Roman;
- размеры полей: левое – 3 см, верхнее – 2 см, правое – 2 см и нижнее – 2 см;
- выравнивание по ширине.
- размер шрифта основного текста – 12;
- интервал межстрочный (полуторный) – 1,5;
- название работы печатается полужирным, размер шрифта – 14;
- заголовки печатаются жирным шрифтом 14-ым размером, перед ними следует оставить пустую строку, выравниваются по центру;
- подзаголовки печатаются жирным шрифтом 12-ым размером выравниваются по центру;
- нумерация страниц – внизу по центру.
- Нумерация рисунков, графиков и т.п. Например: (рис.1 Название рисунка) рисунки нумеруются снизу и по центру, таблица (Таблица 1. Название таблицы) таблицы нумеруются сверху выравнивание к правому краю.
   - Библиографические ссылки при цитировании приводятся в конце статьи и нумеруются согласно порядку цитирования в тексте. Указываются автор (сначала фамилия, потом инициалы), название, место и год издания, страница. Порядковые номера ссылок должны быть написаны внутри квадратных скобок (например: [1], [2]). Источники приводятся с указанием в алфавитном порядке фамилий и инициалов всех авторов, сначала отечественных, затем иностранных, полного названия статьи, названия источника, где напечатана статья, том, номер, страницы (от и до) или полное

название книги, место и год издания. Фамилии иностранных авторов, название и выходные данные их работ даются в оригинальной транскрипции. Каждый источник приводится с новой строки.

9. Материально-техническое обеспечение дисциплины

Минимальные требования к материально-техническому обеспечению дисциплины:
- Компьютерный класс
- проектор, экран
- колонки
- програмное обеспечение NNT Matlab.

10. Глоссарий

ATA – AT-attachment (подключение к шине AT) – интерфейс, использующийся для подключения жестких и оптических дисков к ЭВМ

DIMM– dual in line memory module (модуль памяти с двумя рядами краевых контактов) – один из вариантов конструктивного оформления модулей оперативной памяти, использующийся в основном для синхронной динамической оперативной памяти

DMA – direct memory access (прямой доступ к памяти) – см. *доступ к памяти, прямой*

ICH – input-output (иногда, integrated) control hub (хаб управления вводом-выводом) одна из микросхем системного набора, функционально эквивалентная *южному мосту*, также отличающаяся главным образом средствам связи между соединяемыми компонентами

IDE – integrated device electronics (интегрированная в устройстве электроника) – принцип построения и интерфейс связи с жесткими дисками, при котором большинство функций управления, определяющихся спецификой конкретного жесткого диска, возложено на контроллер, находящийся непосредственно на (или в) корпусе жесткого диска

MCH – memory control hub (хаб управления памятью) – одна из микросхем системного набора, функционально эквивалентная *северному мосту*, но отличающаяся, главным образом, средствами связи между соединяемыми компонентами

PCI – peripheral component interconnect (подключение периферийных компонент) шина расширения, использующаяся для подключения стандартным способом дополнительных устройств к ЭВМ

PCI-Express – развитие шины PCI, обеспечивающее более высокие скорости передачи данных; отличается использованием последовательной передачи данных и связыванием устройств только попарно (точка – точка)

RAID – redundant array of inexpensive disks (избыточный массив недорогих дисков) – технология построения памяти на жестких дисках, использующая несколько дисков для повышения производительности и надежности системы дисковой памяти

RAM – random access memory (память с произвольным доступом) – см. *память с произвольным доступом*; англоязычный термин, часто используемый как синоним оперативного ЗУ

SATA – Serial ATA (последовательный ATA) – интерфейс, использующийся для подключения жестких и оптических дисков к ЭВМ, пришедший на смену обычному ATA (который часто стали называть параллельным), допускающий более высокую скорость передачи и длину соединительного кабеля

SCSI – small computer system interface (интерфейс малых вычислительных систем) – высокоскоростной интерфейс для обмена информацией с несколькими устройствами; часто используется для подключения жестких дисков в серверах

SIMM – single in line memory module (модуль памяти с одним рядом краевых контактов) – один из вариантов конструктивного оформления модулей оперативной памяти, использовавшийся как для асинхронной, так и для синхронной динамической оперативной памяти

Адресации, способ – способ преобразования кода, записанного в адресном поле команды, в виртуальный или физический адрес памяти

АЛУ – см. *арифметико-логическое устройство*

Арифметико-логическое устройство – устройство, обычно часть процессора, непосредственно производящее выполнение операций по переработке информации

Банк памяти – часть запоминающего устройства, имеющая собственные схемы адресации и управления, допускающая возможность обслуживания обращений, адресованных к элементам памяти этой части независимо от остальных частей устройства

Время доступа – одна из основных характеристик запоминающих устройств, показывающая время, необходимое для извлечения информации из ЗУ или записи информации в него; определяется по-разному для различных типов запоминающих устройств

Вычислительная машина, аналоговая – вычислительная машина, использующая аналоговую (модельную) форму представления информации и схемную организацию вычислительного процесса

Вычислительная машина, цифровая – вычислительная машина, использующая алфавитную (дискретную) форму представления информации и программную организацию вычислительного процесса

Доступ, адресный – способ доступа к запоминающему устройству, при котором местоположение извлекаемой или записываемой в ЗУ информации определяется ее адресом (обычно некоторым положительным целым числом)

Доступ, ассоциативный – способ доступа к запоминающему устройству, при котором местоположение извлекаемой или записываемой в ЗУ информации определяется ее значением

Доступ к памяти, прямой – способ организации управления вводом-выводом, при котором внешнее устройство обменивается информацией с оперативной памятью без непосредственного участия процессора

ЗУ – см. *запоминающее устройство*

Запоминающее устройство (ЗУ) – устройство, реализующее функцию хранения информации

Запоминающее устройство, оперативное (ОЗУ) – запоминающее устройство, непосредственно доступное процессору, хранящее программы, выполняемые или готовые к исполнению, и их данные, а также основные программы операционной системы; в англоязычной литературе часто называется памятью с произвольным доступом – *RAM*

Запоминающее устройство, перепрограммируемое (ППЗУ) – запоминающее устройство, как правило, сохраняющее информацию при выключении питания, информация в которое может записываться многократно, причем время записи заметно превосходит время считывания

Запоминающее устройство, постоянное (ПЗУ) – запоминающее устройство, информация в которое заносится однократно (при изготовлении или пользователем) и впоследствии не изменяется, а также сохраняется при выключении питания

Интерфейс – совокупность средств, форматов, правил и протоколов логического и/или физического уровня, определяющих способ взаимодействия между устройствами, программами, функциями, пользователем и информационной системой и т. п.

Канал ввода-вывода – специализированный процессор, предназначенный для управления операциями ввода-вывода

Комплекс, вычислительный – совокупность ЭВМ, каналов связи, периферийного оборудования и других технических средств, а также программного обеспечения и обслуживающего персонала, обычно предназначенный для решения определенного класса задач

Конвейер команд – совокупность основных блоков процессора и памяти, рассматриваемых как последовательность узлов, обслуживающих выполнение команды процессора

Контроллер – универсальное или специализированное устройство управления

Коммутация (в *сети межсоединений*) – способ установления связей и управления передачей информационных пакетов между узлами вычислительной системы или сети

Кэш-память – быстродействующая память (одного или более уровней), располагающаяся между процессором и оперативной памятью; буферная память различных устройств, например жесткого диска

Маршрутизация – выбор пути передачи информации (при наличии нескольких путей) между узлами сети межсоединений вычислительной системы

Массового обслуживания, теория – раздел теории вероятностей, используемый для оценки производительности ЭВМ и систем

Межсоединений, сеть – совокупность средств соединения узлов вычислительной системы

Микропрограммное устройство управления – см. *устройство управления, микропрограммное*

Мост – связь или устройство, использующееся для соединения двух (или более) компонент ЭВМ, системы или сети

Мост, северный - компонент системного набора микросхем для построения ЭВМ, обеспечивающий взаимодействие между процессором, оперативной памятью, видеосистемой и южным мостом

Мост, южный – компонент системного набора микросхем для построения ЭВМ, обеспечивающий взаимодействие с жесткими дисками, средне- и низкоскоростными периферийными устройствами, поддержку шин расширения и включающий в себя ряд контроллеров, например контроллер прерываний, сетевой контроллер, контроллер прямого доступа и др.

фон Неймана, Джона принципы – основные положения, выдвинутые Дж. фон Нейманом, в отношении организации цифровых вычислительных машин

Неймановская архитектура – традиционная архитектура ЭВМ, включающая в себя основные устройства ЭВМ и отвечающая принципам Дж. фон Неймана

Неупорядоченное исполнение команд – механизм исполнения команд программы в порядке, отличном от их следования в программе, обеспечивающий в результате правильную логическую последовательность выполнения программы

Организация памяти, сегментная – механизм представления памяти ЭВМ в виде совокупности логических блоков различной длины и отображения этой совокупности на запоминающие устройства, входящие в состав ЭВМ

Организация памяти, страничная – механизм разбиения памяти ЭВМ на блоки постоянной длины, использующиеся для обмена между ступенями памяти; служит (часто вместе с *сегментной организацией*) основой для построения *виртуальной памяти*

Памяти, тайминги – временн***ы***е параметры оперативных запоминающих устройств ЭВМ, характеризующие время выполнения отдельных этапов цикла обращения

Памяти, уровень – совокупность запоминающих устройств ЭВМ или системы, объединенных общим функциональным назначением и обладающих близкими характеристиками

Память, асинхронная – запоминающее устройство, сигналы цикла обращения к которому не синхронизируются

Память, виртуальная – совокупность запоминающих устройств ЭВМ, рассматриваемая логически как некоторое логически однородное пространство адресов памяти (или средства реализации такого представления)

Память, оперативная – см. *запоминающее устройство, оперативное*

Память, сверхоперативная – быстродействующее запоминающее устройство, применяющееся для ускорения обмена информацией между процессором и оперативной памятью; в настоящее время для обозначения таких устройств чаще используется термин *кэш-память*

Память, синхронная – запоминающее устройство, сигналы цикла обращения к которому синхронизируются специальной последовательностью сигналов синхронизации

Память с произвольным доступом – запоминающее устройство с адресным доступом, выбор элементов которого при записи или чтении производится посредством системы внутренних линий (обычно линий строк и столбцов), выбираемых дешифратором в соответствии с заданным адресом

Переходов предсказание – (предположительный) выбор ветви программы в разветвлении по условию до определения значения условия

Периферийное устройство – устройство, включенное в состав ЭВМ, комплекса или системы, но не входящее непосредственно в состав центрального или периферийных процессоров, системного набора микросхем и каналов ввода-вывода

Прерывание – процедура прекращения выполнения текущей программы при возникновении определенных ситуаций в ЭВМ или системе, сопровождаемая переходом к программе обработки возникшей ситуации и предполагающая возможность последующего возврата к прерванной программе

Прерываний, контроллер – контроллер, обеспечивающий прием сигналов запросов прерываний, их предварительную обработку (маскирование, приоритетный выбор) и передачу в процессор

Производительность ЭВМ – характеристика ЭВМ, отражающая затраты времени ЭВМ на решение задач; может измеряться количеством задач, решаемых в единицу времени (пропускная способность)

Процессор – центральный блок ЭВМ, осуществляющий основные действия по переработке информации, а также управлению ЭВМ в целом

Процессор, векторный – процессор, позволяющий выполнять одновременную обработку нескольких скалярных величин (обычно компонент вектора)

Процессор, **суперскалярный** – процессор, имеющий несколько исполнительных блоков и как правило, более одного конвейера команд

Процессора, ядро – часть процессора, обеспечивающая обработку последовательности команд, обычно включающая в себя исполнительные блоки, управляющую часть и регистры; процессор может иметь одно или несколько ядер

Прямой доступ к памяти – см. *доступ к памяти, прямой*

Расслоение обращений к памяти – способ назначения адресов ячеек (байтов) в многоблочной памяти, при котором последовательные адреса размещаются в смежных блоках памяти

Регенерация информации в динамических ЗУ – периодически выполняемая процедура восстановления записанной в ЗУ информации, необходимость которой вызвана самопроизвольным разрядом запоминающих конденсаторов

Режим работы ЭВМ – порядок выполнения программ в ЭВМ и особенности его взаимодействия с пользователями; различают однопрограммный и многопрограммные режимы, режимы коллективного доступа, реального времени, пакетной обработки

Северный мост – см. *мост, северный*

Сервер – ЭВМ, имеющая специальное функциональное назначение или структуру, ориентированную на обеспечение специальных требований

Сеть, вычислительная – совокупность вычислительных машин, соединенных между собой каналами связи, как правило, разнесенных территориально

Сеть, межсоединений – см. *межсоединений, сеть*

Система, вычислительная – совокупность устройств, содержащих один или более процессоров (или ЭВМ), запоминающих и периферийных устройств, объединенных шинами расширения или каналами связи

Способ организации вычислительного процесса – способ задания последовательности действий, обеспечивающих решение задачи на ЭВМ; различают алгоритмическую (программную) и схемную организацию вычислительного процесса

Суперскалярный процессор – см. *процессор, суперскалярный*

Схемное устройство управления – см. *устройство управления, схемное*

Устройство управления (УУ) – устройство, вырабатывающее последовательность управляющих сигналов для операционных блоков в соответствии с заданным алгоритмом управления

Устройство управлениямикропрограммное – устройство управления, последовательность микрокоманд в котором записана в постоянной памяти

Устройство управления, схемное – устройство управления, последовательность микрокоманд в котором формируется с помощью управляющего автомата, состоящего из памяти состояний и комбинационной схемы

Форма представления информации – вид отображения информации, используемый в вычислительной машине; две основные формы: алфавитная (цифровая) и аналоговая

Шина расширения – внутренняя шина ЭВМ, используемая для подключения к ней дополнительных устройств, например, шина *PCI*

Эмуляция – воспроизведение программными или аппаратными средствами одной ЭВМ или устройства функционального поведения другой ЭВМ или устройства

Эффективность ЭВМ – критерий, характеризующий степень соответствия ЭВМ своему назначению, или в простом случае соотношение затрат и производительности ЭВМ

Приложения

ПРИЛОЖЕНИЕ 1

Лекция 1.

Современные ЭВМ строятся на одном принципе – принципе программного управления. В основе принципа программного управления лежит представление алгоритма в форме операторной схемы, которая задает правило вычислений, как композицию операторов (операций над информацией) двух типов: операторов, обеспечивающих преобразование информации, и операторов, анализирующих информацию с целью определения порядка выполнения операторов. Реализация этого принципа в различных ЭВМ может быть разной. Используемый в современных компьютерах принцип программного управления был предложен в 1945 году Дж. фон Нейманом, и с тех пор неймановский принцип программного управления используется в качестве основного принципа построения ЭВМ. Этот принцип включает следующие положения.

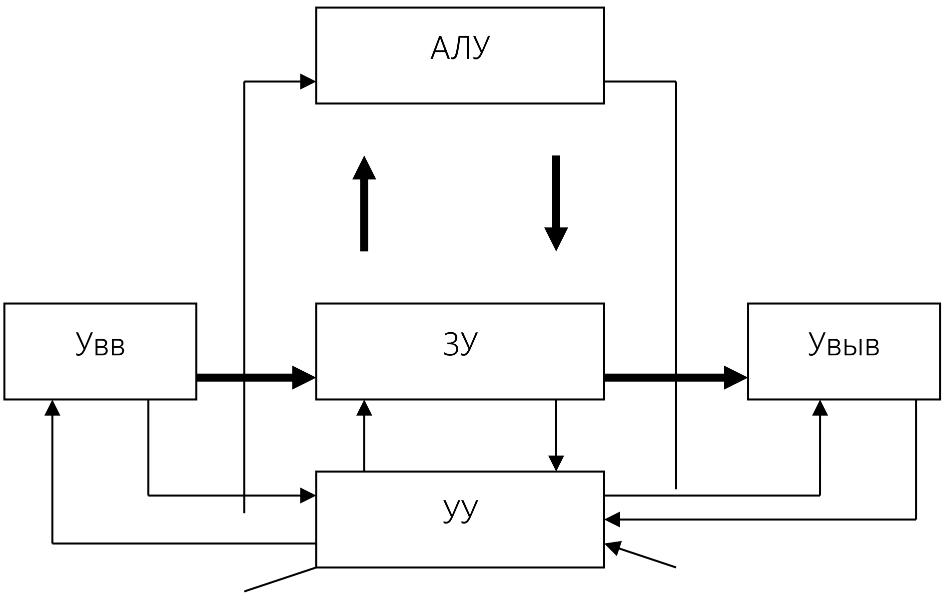
1. Информация кодируется в двоичной форме и разделяется на единицы (элементы) информации, называемые *словами*.
2. Разнотипные слова информации (числа, символы, команды) различаются по способу использования, но не способами кодирования.
3. Слова информации размещаются в ячейках памяти машины и идентифицируются номерами ячеек, называемыми *адресами слов.*
4. Алгоритм представляется в форме последовательности управляющих слов, которые определяют наименование операции и слова информации, участвующие в операции, и называются *командами*. Алгоритм, представленный в терминах машинных команд, называется *программой*.
5. Выполнение вычислений, предписанных алгоритмом, сводится к последовательному выполнению команд в порядке, однозначно определяемом алгоритмом.

Классическая схема ЭВМ.

Изложенные принципы программного управления (фон Неймана) реализовывались в аппаратном обеспечении, структура которого постепенно оформилась а приведенную ниже схему, ставшую к настоящему времени классической и включающую:

- блок для выполнения логических и арифметических операций (АЛУ);
- блок для хранения информации (память) или ОЗУ;
- устройства для ввода и вывода данных.

Для обеспечения согласованной работы вышеперечисленных устройств преобразования информации требуется устройство управления (УУ).



На этой схеме линиями разной толщины отмечены потоки: информации
управляющих сигналов

Кратко функционирование устройств ЭВМ можно описать так.

УУ инициирует работу Увв, давая ему команду на выполнение ввода в ЗУ, аналогично инициируется работа Увыв.

УУ указывает, из какой ячейки памяти ЗУ необходимо передавать информацию в АЛУ, какую операцию над этой информацией должно выполнять АЛУ, в какую ячейку памяти должен быть занесен результат операции.

Современные ЭВМ имеют отличия, обусловленные развитием ВТ.

Основные отличия:

- ЗУ представляется несколькими уровнями: Внутреннее или оперативное ЗУ (ОЗУ) и внешнее или ВЗУ. Внутреннее ЗУ содержит информацию, обрабатываемую в определенный промежуток времени, включающий и текущий момент. Внешние ЗУ служит хранилищем всей информации для конкретного пользователя. В современной ЭВМ внешние ЗУ насчитывают несколько уровней.
- АЛУ и УУ объединены в одно устройство, называемое центральным процессором (ЦП);
- в современных ЭВМ и ПК имеется довольно большой арсенал Увв и Увыв.

Структурно современные. ЭВМ и ПК состоит из 2-х частей: центральной и периферийной. К центральной части относят процессор и ОЗУ.

ЦП называют устройство, непосредственно осуществляющее процесс обработки данных и программное управление этим процессом. В его состав входят АЛУ, УУ, и собственная память процессора. В современных ПК, ЦП реализован в виде большой интегральной схемы и называется микропроцессором.

ЦП взаимодействует с ОЗУ или просто оперативной памятью (ОП). ОП предназначена для приема, хранения и выдачи информации (чисел, символов, команд, констант) – всей информации для выполнения вычислений по программе.

Кроме ОП во всех ПК имеется внутренняя постоянная память для хранения постоянных данных и программ. ОП - дорогая часть аппаратуры. Объем ее ограничен, поэтому большие массивы или таблицы информации хранятся в ВЗУ. К ним относят: накопители на магнитных дисках (НМД), накопители на магнитных лентах (НМЛ), накопители на оптических и магнитооптических дисках.

В современных ПК реализована виртуальная память, которая позволяет пользователю работать с расширенным пространством памяти компьютера. Виртуальная память представляет совокупность ОП и ВЗУ, а также комплекса программно-аппаратных средств, обеспечивающих динамическую переадресацию данных, в результате чего пользователь не заботится о своевременной передаче информации из ВЗУ в ОП. Функции по требуемому перемещению берет на себя вычислительная система.

ВЗУ, Увв и Увыв образуют периферийную часть ПК. Состав устройств может сильно отличаться в разных экземплярах, поэтому говорят о конфигурации ПК.

Производительность и эффективность использования ПК определяется не только процессором и ОП, но в большой степени и техническими данными периферийных устройств, а также способом их совместной работы с центральной частью ПК. Связь между центральной и периферийной частями осуществляется с помощью сопряжений, которые называются *интерфейсами.*

Интерфейс – совокупность стандартизованных аппаратных и программных средств, обеспечивающих обмен информации между устройствами. В основе построения интерфейсов лежит унификация и стандартизация (использование единых способов кодирования данных, форматов данных, использование единых разъемов и т.п.). Наличие интерфейсов позволяет унифицировать передачу данных между устройствами независимо от их особенностей. Особенности учитывают *контроллеры* – устройства управления периферийных устройств.

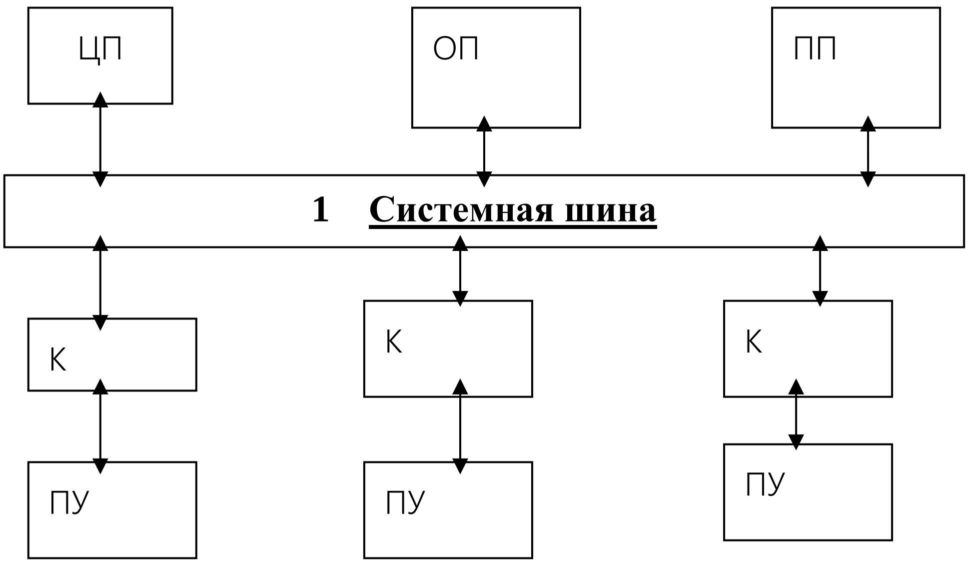
В настоящее время в ПК используется структура с одним общим интерфейсом, называемым *системной шиной*. При такой структуре все устройства ПК обмениваются информацией и управляющими сигналами через системную шину. Физически она представляет собой систему функционально объединенных проводов, по которым передается 3 потока данных:

- непосредственно информация (данные);
- управляющие сигналы;
- адреса.

Ниже приведена схема обмена информацией между устройствами в ЭВМ с общей шиной.

Количество проводов в системной шине, предназначенных для передачи информации называется *разрядностью шины*.

Поскольку шина является общей, использоваться она может в каждый определенный момент времени только одним каким-либо устройством. Для этих целей предусмотрена система приоритетных прерываний, которая отдает шину для использования устройству с наибольшим приоритетом.



ЦП – процессор, ОП –память, ПП –постоянная память, К – контроллер, ПУ- периф. устр-во

Лекция 2.

Классификация ЭВМ

ЭВМ могут быть классифицированы по ряду признаков, например, по принципу действия, по этапам создания и по элементной базе, назначению, способу организации вычислительного процесса, размеру и вычислительной мощности, функциональным возможностям и другим признакам.

Рассмотрим некоторые классификационные группировки.

Прежде всего не надо забывать, что помимо цифровых ЭВМ, существуют аналоговые и гибридные ЭВМ. В основе определения этих классификационных групп лежит такой признак классификации, как принцип действия. В этой классификации критерием разделения на группы является форма представления информации. В цифровых ЭВМ информация представляется дискретными сигналами-импульсами, представляющими коды 0 и 1. В аналоговых ЭВМ работают с информацией, представленной в непрерывной (аналоговой) форме, а гибридные ЭВМ, как следует из их названия, могут работать как с цифровыми, так и с аналоговыми сигналами. Следует отметить, что большая часть классификации относится к цифровым вычислительным машинам, хотя определение "цифровые" опускается.

Наиболее известна классификация цифровых ЭВМ по этапам создания и элементной базе, согласно которой они условно делятся на поколения:

- 1-е поколение, 50-е годы: ЭВМ на электронных вакуумных лампах (термин "электронные" является частью названия этой элементной базы);
- 2-е поколение, 60-е годы: ЭВМ на полупроводниковых приборах (транзисторах);
- 3-е поколение, 70-е годы: ЭВМ на полупроводниковых интегральных схемах с малой и средней степенью интеграции;
- 4-е поколение, 80-90-е годы: ЭВМ на больших и сверхбольших интегральных схемах, основная из которых – микропроцессор (сотни тысяч – десятки миллионов активных элементов водном кристале);
- 5-е поколение, настоящее время: компьютеры с многими десятками параллельно работающих микропроцессоров;
- 6-е поколение: оптоэлектронные компьютеры с массовым параллелизмом и нейронной структурой.

По назначению ЭВМ делятся на универсальные (общего назначения), проблемно-ориентированные и специализированные. Универсальные ЭВМ предназначены для решения самых разных инженерно-технических, экономических, математических, информационных и других задач. Их характерными чертами являются:

- высокая производительность;
- разнообразие форм обрабатываемых данных: двоичных, десятичных, символьных – и большие диапазоны их изменения наряду с высокой точностью представления;
- обширная номенклатура выполняемых операций, как арифметических и логических, так и специальных;
- большая емкость оперативной памяти;
- развитая организация системы ввода-вывода информации, обеспечивающая подключение разнообразных видов внешних устройств.

Проблемно-ориентированные ЭВМ предназначены для решения достаточно узкого класса задач, связанных с управлением технологическими объектами; регистрацией, накоплением и обработкой небольшого объема данных; выполнением расчетов по несложным алгоритмам. Они обладают по сравнению с универсальными ЭВМ ограниченными аппаратными и программными ресурсами при сравнительно невысокой стоимости. Специализированные ЭВМ предназначены для решения определенного узкого класса задач или реализации строго определенной группы функций. Узкая ориентация позволяет четко специализировать их структуру, существенно снизить их сложность и стоимость при сохранении высокой производительности и надежности. К таким ЭВМ относятся программируемые микропроцессоры специального назначения, адаптеры и контроллеры, выполняющие логические функции управления отдельными несложными техническими устройствами, агрегатами и процессами.

По размерам и вычислительной мощности ЭВМ делят на сверхбольшие (суперЭВМ), большие, малые, сверхмалые (микроЭВМ). В таблице 2.2 приводятся основные технические характеристики ЭВМ разных классов.

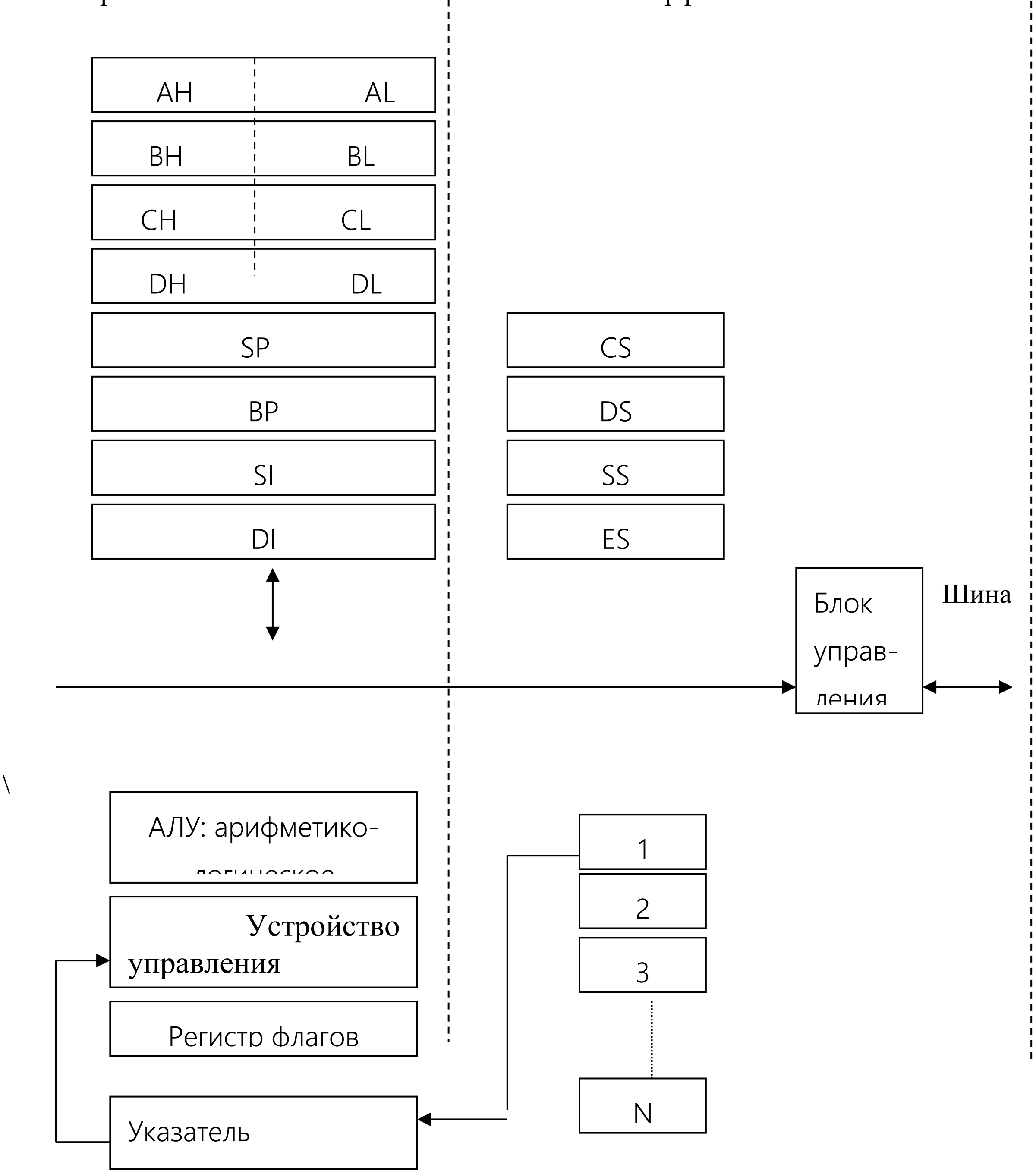
Таблица 2.2.

Техн. характеристики	СуперЭВМ	Большие ЭВМ	Малые ЭВМ	МикроЭВМ
Производительность, MIPS	1000-1000000	100- 10000	10-1000	10-100
Емкость ОП, Мбайт	2000-100000	512-10000	128-2048	32-512
Емкость ВЗУ, Гбайт	500-50000	100-10000	20-500	10-50
Разрядность шины, бит	64-256	64-128	32-128	32-128

Лекция 3.

Центральный процессор

УИ: Устройство исполнения ШИ: ~~Шинный интерфейс~~



Как показано на рис. выше, процессор делится на 2 логических устройства: устройство исполнения (УИ) и шинный интерфейс (ШИ). УИ ответственно за выполнение инструкций, а ШИ – за доставку УИ данных и инструкций для обработки. УИ содержит АЛУ , УУ и регистры.

Регистрами в схемотехнике называют электронные устройства, в которые можно записать информацию, сохранить ее и, при необходимости, прочитать. При этом регистры, как правило, обеспечивают более быстрый доступ к информации, чем доступ к содержимому ячеек памяти ОЗУ. Это обстоятельство и является причиной оснащения центрального процессора ЭВМ собственной памятью, причем в более поздних разработках микропроцессоров собственная память процессоров только увеличивается. Intel 8086 имеет 14 регистров:

- универсальные AX, BX, CX, DX;
- сегментные регистры CS, DS, SS, ES;
- регистры смещения IP, SP, BP, SI, DI;
- регистр флагов (регистр состояния) FL.

Процессор содержит двенадцать 16-разрядных программно-адресуемых регистров, которые принято объединять в три группы: регистры данных, регистры-указатели и сегментные регистры. Кроме того, в состав процессора входят счетчик команд и регистр флагов (рис. 3.1).

1. Регистр-аккумулятор (Accumulator register) EAX/AX/AH/AL применяется для хранения промежуточных данных, в некоторых командах его использование обязательно;

2. Базовый регистр (Base register) EBX/BX/BH/BL применяется для хранения базового адреса некоторого объекта в памяти;

3. Регистр-счетчик (Count register) ECX/CX/CH/CL применяется в командах, производящих некоторые повторяющиеся действия. Использование регистра-счетчика зачастую скрыто в алгоритме работы той или иной команды. Например, команда организации цикла LOOP помимо передачи управления анализирует и уменьшает на единицу значение регистра ECX/CX;

4. Регистр данных (Data register) EDX/DX/DH/DL, так же как и регистр EAX/AX/AH/AL, хранит промежуточные данные.

Для операций с массивами:

1. регистр индекса источника (Source Index register) ESI/SI содержит текущий адрес элемента в массиве-источнике;

2. регистр индекса приемника (Destination Index register) EDI/DI то же, но в массиве-приемнике.

Для операций со стеками:

1. регистр указателя стека (Stack Pointer register) ESP/SP содержит указатель на вершину стека в текущем сегменте стека;

2. регистр указателя базы кадра стека (Base Pointer register) EBP/BP предназначен для организации произвольного доступа к данным внутри стека.

Любая программа состоит из кода, данных и стека. В IA32 шесть сегментных регистров CS (сюда код), SS (сюда стек), DS (сюда данные), ES. Указатель команд IP «следит» за ходом выполнения программы, указывая в каждый момент относительный адрес команды, следующей за исполняемой. Регистр IP программно недоступен (IP — это просто его сокращенное название, а не мнемоническое обозначение, используемое в языке программирования); наращивание адреса в нем выполняет микропроцессор, учитывая при этом длину текущей команды.

Регистр флагов, эквивалентный регистру состояния процессора других вычислительных систем, содержит информацию о текущем состоянии процессора (рис. 3.2). Он включает 6 флагов состояния и 3 бита управления состоянием процессора, которые, впрочем, тоже обычно называются флагами.

Биты

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
				OF	DF	IF	TF	SF	ZF		AF		PF		CF

Регистр флагов

Флаг переноса CF (Carry Flag) индицирует перенос или заем при выполнении арифметических операций, а также служит индикатором ошибки при обращении к системным функциям.

Флаг паритета PF (Parity Flag) устанавливается в 1, если младшие 8 бит результата операции содержат четное число двоичных единиц.

Флаг вспомогательного переноса AF (Auxiliary Flag) используется в операциях над упакованными двоично-десятичными числами. Он индицирует перенос или заем из старшей тетрады (бита 4).

Флаг нуля ZF (Zero Flag) устанавливается в 1, если результат операции равен нулю.

Флаг знака SF (Sign Flag) показывает знак результата операции, устанавливаясь в 1 при отрицательном результате.

Флаг переполнения OF (Overflow Flag) фиксирует переполнение, т.е. выход результата операции за пределы допустимого для данного процессора диапазона значений.

Флаги состояния автоматически устанавливаются процессором после выполнения каждой команды. Так, если в регистре AX содержится число 1, то после выполнения команды декремента (уменьшения на 1)

```
dec  AX
```

содержимое AX станет равно 0, и процессор сразу отметит этот факт, установив в регистре флагов бит ZF (флаг нуля). Если попытаться сложить два больших числа, например, 58000 и 61000, то установится флаг переноса CF, так как число 119000, получающееся в результате сложения, должно занять больше двоичных разрядов, чем помещается в регистрах или ячейках памяти, и возникает «перенос» старшего бита этого числа в бит CF регистра флагов.

Индицирующие флаги процессора дают возможность проанализировать, если это нужно, результат последней операции и осуществить «разветвление» программы: например, в случае нулевого результата перейти на выполнение одного фрагмента программы, а в случае ненулевого — на выполнение другого фрагмента. Такие разветвления осуществляются с помощью команд условных переходов, которые в процессе своего выполнения анализируют состояние регистра флагов. Так, команда

```
jz   zero
```

осуществляет переход на метку zero, если результат выполнения предыдущей команды окажется равен нулю (т.е. флаг ZF установлен), а команда

```
jnc  okey
```

выполнит переход на метку okey, если предыдущая команда сбросила флаг переноса CF (или оставила его в сброшенном состоянии).

Управляющий флаг трассировки TF (Trace Flag) используется в отладчиках для осуществления пошагового выполнения программы. Если TF = 1, то после выполнения каждой команды процессор реализует процедуру прерывания 1 (через вектор прерывания с номером 1).

Управляющий флаг разрешения прерываний IF (Interrupt Flag) разрешает (если равен 1) или запрещает (если равен 0) процессору реагировать на прерывания от внешних устройств.

Управляющий флаг направления DF (Direction Flag) используется особой группой команд, предназначенных для обработки строк. Если DF = 0, строка обрабатывается в прямом направлении, от меньших адресов к большим; если DF = 1, обработка строки идет в обратном направлении.

Основная функция ШИ – управление шиной, сегментными регистрами и очередью исполнения. ШИ управляет шинами, передающими данные в УИ, память и внешние устройства ввода/вывода. Сегментные регистры управляют адресацией памяти.

Еще одна функция ШИ – предоставлять доступ к инструкциям (командам программы). Поскольку инструкции исполняемой программы находятся в памяти, ШИ должен получить к ним доступ и поместить в очередь исполнения, размер которой меняется в зависимости от типа процессора и определяется размером кэша команд. Это позволяет ШИ предсказывать и загружать необходимые инструкции заранее, так что в очереди всегда есть инструкции, готовые к исполнению.

УИ и ШИ работают параллельно, и ШИ всегда на 1 шаг опережает УИ. УИ сообщает ШИ, когда ему необходим доступ к данным в памяти или к устройствам ввода/вывода. УИ также запрашивает инструкции для исполнения из очереди в ШИ. Самая верхняя инструкция – исполняемая в данный момент, а пока УИ занято выполнением данной инструкции, ШИ извлекает следующую инструкцию из памяти. Выполнение инструкций и их получение из памяти перекрываются по времени, за счет чего растет быстродействие процессора. (Заметим, что программисты не имеют доступа к вышеописанным элементам процессора).

Лекция 4.

Внутренняя память

Внутренняя память ПК подразделяется на постоянную (ПЗУ) и оперативную (ОЗУ). Минимальной адресуемой единицей памяти является байт. Байты в памяти нумеруются последовательно, начиная с 0, и каждый байт имеет свой уникальный номер-адрес. Структура адресного пространства памяти ПК на примере PC Intel 8086 показана ниже.

Таблица – Структура адресного пространства внутренней памяти ПК

Начало (десятичный номер килобайта)	Адрес (шестнадца-теричный	Объем в килобайтах	Назначение
0	00000	640	Основная оперативная память, включающая Таблицу векторов прерывания Область BIOS
640	A0000	128	Видеопамять (ОЗУ)
768	C0000	192	Расширенная область (ПЗУ)
960	F0000	64	Основное ПЗУ системы

Для непосредственного использования прикладными программами доступна большая часть основной оперативной памяти. ПЗУ состоит из специальных чипов, информация из которых, будучи однажды записана, в дальнейшем только считывается. В силу этого свойства, записанные в таких чипах данные и инструкции не могут быть изменены (в английской транскрипции ПЗУ называется ROM – Read Only Memory, что означает “память только для чтения”). Расширенное ПЗУ включает постоянную BIOS и отвечает за устройства ввода/вывода, такие, как контроллер (устройство управления) винчестера и др. Основное ПЗУ системы обеспечивает самопроверку при включении компьютера, загрузку операционной системы с диска, рисование точек при выводе графики. Когда Вы включаете компьютер, ПЗУ управляет различными проверками и загрузкой специальных данных с диска в ОЗУ (заметим, что для программиста эта память недоступна).

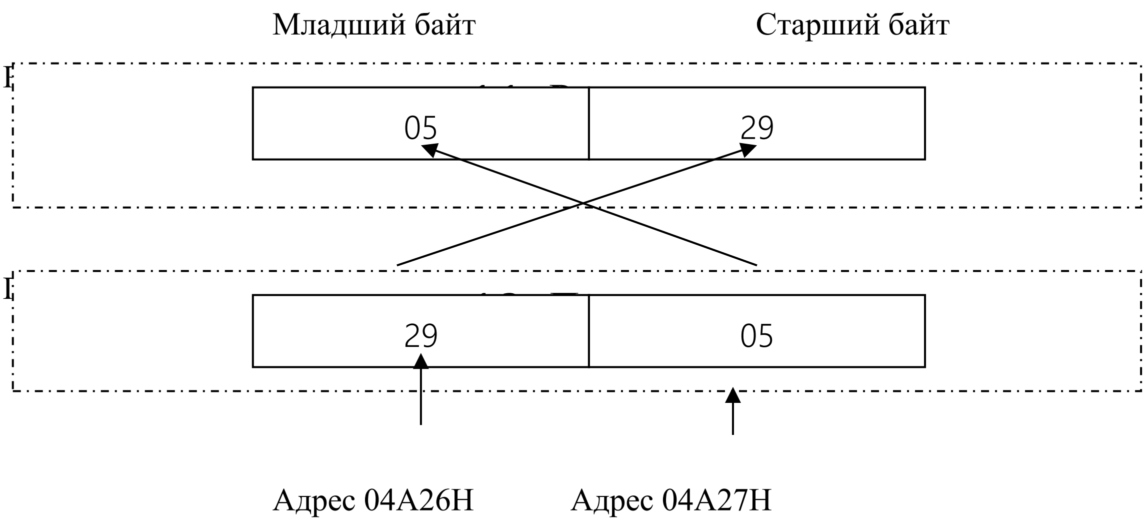
ОЗУ доступна программисту в качестве пространства для временного хранения данных и выполнения программ. При включении компьютера часть операционной системы загружается с винчестера в оперативную память. Остальная память из основного ОЗУ может использоваться программистом. Выполняемая прикладная программа находится в этой части ОЗУ и выводит результаты обработки на экран дисплея, принтер или устройства внешней памяти (например, НГМД или винчестер). После завершения одной прикладной программы операционная система может загрузить на ее место другую программу.

Выключение компьютера приводит к потере данных в ОЗУ, но не влияет на ПЗУ. Вот почему требуемые для дальнейшей обработки данные необходимо сохранять в долговременной памяти.

Сегментация и адресация

Процессор может обращаться к одному или более байтам памяти. Под словом в процессоре понимается двухбайтовая ячейка памяти. Для уяснения вопроса о размещении данных в ОЗУ, рассмотрим десятичное число 1315, шестнадцатеричный эквивалент которого равен 0529H. Это значение может быть помещено в слово в ОЗУ. Оно состоит из старшего байта, значение которого равно 05, и младшего байта со значением 29 *(привыкайте, что значения в памяти отображаются в шестнадцатеричном виде).* Процессор хранит данные в памяти в инверсном виде: младший байт – по младшему адресу, а старший – по старшему, на 1 больше предыдущего. Поскольку дамп памяти (отображение содержимого ячеек памяти) представляется отладчиками (TD, DEBUG) в порядке увеличения адресов байтов, интерпретировать значения слов или двойных слов надо справа налево. Например, значение 0529H в памяти компьютера, начиная с адреса 04A26H, будет храниться следующим образом: значение 29H в байте с адресом 04A26H, а 05 – 04A27H. Нижеприведенный рисунок иллюстрирует размещение слова в памяти и в регистре.

При программировании на ассемблере надо ясно представлять себе разницу между адресом ячейки памяти и ее значением. В рассматриваемом нами примере ячейка, имеющая адрес 04A26, хранит значение 29, а ячейка с адресом 04A27 – 05. В описаниях работы и в алгоритмах обработки информации часто адрес называется указателем переменной, а содержимое указателя – значением переменной..



Расположение байтов слова в памяти компьютера и в регистре

Сегменты – это специальные области, определяемые в программе для хранения разных функциональных частей программы: кода программы (алгоритма обработки данных), переменных программы (значений данных), рабочих ячеек для временного хранения промежуточных значений (в программах такая структура называется *стеком*). Максимальный размер сегмента в реальном режиме может достигать 64K байт, но в каждой программе сегмент занимает столько места в ОЗУ, сколько требуется для размещения команд программы или обрабатываемых программой данных. Начало каждого сегмента (адрес, в который заносится первая команда или первая описываемая переменная) фиксируется (записывается) в сегментном регистре. В реальном режиме используются 3 основные сегмента и соответствующих сегментных регистра: кодовый – CS, данных – DS, стека – SS.

Сегментный регистр CS содержит адрес инструкции, к которой обращается ОС для начала выполнения программы. Сегментный регистр DS адресует область переменных и ячеек, зарезервированных для результатов обработки данных. Сегментный регистр SS содержит адрес специальной структуры для временного сохранения данных или данных, используемых программой в собственных “вызываемых” подпрограммах. Сегмент начинается с границы параграфа, т.е. с адреса, кратного 16 (10H). Поскольку для всех адресов, делящихся нацело на 16, младший шестнадцатеричный разряд равен 0, разработчики компьютера решили не хранить этот разряд в сегментном регистре, уменьшив за счет этого размер регистра на 4 бита. Поэтому адрес сегмента 038E0H будет хранится в сегментном регистре как 038EH. При необходимости ссылки на физический (фактический) адрес начала сегмента, такой адрес будем записывать в виде 038E[0]H.

В программе все ячейки памяти в сегменте нумеруются относительно адреса в соответствующем сегментном регистре. Количество байт от начала сегмента до любого адресуемого в сегменте байта называется *смещением (offset).* В реальном режиме работы смещение в сегменте размером 64К байт должно изменяться от 0000H до FFFFH, т.е. для указания смещения в сегменте достаточно 16 двоичных разрядов. Т.о., имея 2 16-битовых регистра для адреса начала сегмента и смещения внутри сегмента, можно адресовать память в 1М байт (не имея сегментации потребовалось бы для адресации ячеек такой памяти 20 двоичных разрядов!). Чтобы обратиться к любой ячейке памяти в сегменте, процессор определяет *физический* адрес ячейки, складывая значения в сегментном регистре со смещением, при этом не забывая восстановить “отрезанный” шестнадцатеричный нуль младшего разряда сегментного регистра. Например, если в сегменте с начальным адресом 038E[0]H команда программы имеет смещение 0032H, то физический адрес такой команды в ОЗУ будет 038E[0]H+0032H=03912H. Эту операцию определения физического адреса в ОЗУ процессор выполняет автоматически.

Итак, в компьютере существуют 2 основные схемы адресации:

1) *абсолютный или физический* адрес, представляющий собой 20-разрядное число, прямо указывающее на определенную ячейку ОЗУ;
2) адрес в системе *сегмент:смещение*, состоящий из начального адреса сегмента и значения смещения. Каждая составляющая *сегмент:смещение* представляет собой 16-разрядное число.

(Аналогом абсолютной адресации была бы последовательная нумерация всех зданий в городе, а аналогом системы *сегмент:смещение* является определение адреса по улице и номеру дома на этой улице).

Программисты редко имеют дело с абсолютной адресацией и даже редко обращают внимание на начало сегмента (ОС сама в большинстве случаев записывает в сегментный регистр соответствующие адреса), а вот смещение внутри сегмента часто приходится учитывать в программах обработки.

1. Простейшая программа Hello World!

```
model small
.stack 256
.data
	strHelloWorld DB 'Hello World!$'
.code
start:
	mov ax, @data
	mov ds, ax
	mov dx, offset  strHelloWorld
	mov ah, 09h
	int 21h
	mov ah, 04Ch
	int 21h
end start
```

2. Ввод и вывод символа на экран

```
model small
.stack 256
.data
strNewLine db 0ah, 0dh, 024h
strInput  db 'Intput the symbol: $'
strOutput db 'Your enter: $'
symbol db ?
dol db '$'
```

```
.code
main:
  mov ax, @data
  mov ds, ax

  mov ah, 09h
  mov dx, offset strInput       ; выводим строку strInput
  int 21h

  mov ah, 1h            ; заносим номер функции в необходимый регистр
  int 21h               ; ввод символа
  mov symbol, al        ; код символа из AL помещаем в symbol

  mov dx, offset strNewLine ; в strNewLine находится символ перехода на новую строку
(0ah),
  mov ah, 09h                   ; перехода на начало строки (0dh)
  int 21h                               ; и символ конца строки '$' (024h)

  mov dx, offset strOutput      ; выводим строку strOutput
  int 21h

  mov dx, offset symbol                 ; в DX помещаем смещение переменной symbol
  int 21h                       ; для вывода значения из symbol

  mov ah, 04ch          ; выход
  int 21h

end main                ; конец программы
```

3. Ввод и вывод строки на экран

```
model small
.stack 256
.data
bufferSize db 10        ; 1 байт - размер буфера для ввода
bufferLength db ?       ; 2 байт - число фактически введенных символов
strInput db 20 dup(?)  ; 3 байт - введенная строка  с символом 0dh на конце
strNewLine db 0ah, 0dh, 024h
strInputMsg db 'Intput the string: $'
strOutputMsg db 'Your enter: $'

.code
main:

   mov ax, @data
   mov ds, ax

   mov ah, 09h
   mov dx, offset strInputMsg           ; выводим строку strInput
   int 21h
```

```
    mov ah, 0ah                ; определяем параметры для нашей функции
    mov dx, offset bufferSize
    int 21h

    xor bx, bx                 ; очищаем регистр bx
    mov bl, bufferLength
    add bl, 2
    mov ah, 024h
    mov [bx], ah

    mov ah, 09h
    mov dx, offset strNewLine        ; переходим на новую строку
    int 21h
    mov dx, offset strOutputMsg
    int 21h
    mov dx, offset strInput    ; выводим введенную строку
    int 21h

    mov ax, 4c00h
    int 21h
end  main
```

Лекция 5.

Пересылки. Арифметические команды.

Местонахождение операнда	Обозначение	Запись в ЯА
в команде	i8, i16, i32	константное выражение
в регистре общего назначения	r8, r16	имя регистра
в сегментном регистре	sr	CS, DS, SS, ES
в ячейке памяти	m8, m16, m32	адресное выражение

Команды пересылки.

Команда MOV – на место первого операнда пересылает значение второго операнда op1:=op2. Флаги не меняет. Синтаксис: mov op1, op2

op1	*op2*	
r8 m8	i8, r8, m8 i8, r8	Пересылка байтов
r16 sr (кроме cs) m16	i16, r16, sr, m16 r16, m16 i16, r16, sr	Пересылка слов

Примеры:

```
X DB ?              ; TYPE X = BYTE
Y DW ?              ; TYPE Y = WORD
MOV BH, 0   ; пересылка байта
MOV  X, 0
```

MOV SI, 0 ; пересылка слова (SI – регистр размеров в слово)
MOV Y, 0

Если можно определить размеры обоих операндов, то эти размеры должны совпадать, иначе ассемблер зафиксирует ошибку:

MOV DI, ES ; пересылка байта
MOV CH, X ; пересылка слова
MOV DX, AL ; ошибка (DX – слово, AL - байт)
MOV BH, 300 ; ошибка (BX – байт, а 300 не может быть байтом)

В случае, когда по операндам команды MOV нельзя определить размер пересылаемой величины, используется оператор указания типа PTR.

MOV [SI], 0 ; в регистре SI находится адрес ячейки памяти и
; требуется записать 0 в эту ячейку (исполнит. адрес)

Синтаксис: <тип> PTR <выражение>
<тип> - это BYTE, WORD, DWORD, а выражение может быть константным или адресным
Обнуление байта по адресу SI:

MOV BYTE PTR [SI], 0 ; или MOV [SI], BYTE PTR 0

Обнуление слова по адресу SI:

MOV WORD PTR [SI], 0 ; или MOV [SI], WORD PTR 0

Используется также, когда требуется не уточнить тип операнда, а изменить его.
Например:

Z DW 1234h
MOV Z, 0 ; Z: 00h, Z+1: 00h
MOV BYTE PTR Z, 0 ; Z: 00h, Z+1: 12h
MOV BYTE PTR (Z+1), 0 ; Z: 34h, Z+1: 00h

Итак, оператор PTR используется в следующих ситуациях: когда типы операндов команд неизвестны и потому надо указать тип одного из операторов, и когда не устраивает тип, приписанный имени при его описании, и потому мы должны указать нужный тип.

Команда XCHG.

Перестановка (exchange): XCHG op1, op2

Меняет местами значения своих операндов (они должны быть либо байтами, либо словами): op1 ⇔ op2. Флаги при этом не меняются

op1	*op2*	
r8	r8, m8	перестановка байтов
m8	r8	
r16	r16, m16	перестановка слов
m16	r16	

Пример:

MOV AX, 62 ; AX=62
MOV SI, 135 ; SI=135
XCHG AX, SI ; AX=135, SI=62

Команды сложения и вычитания.

Для беззнаковых чисел:

$$\text{сумма}(x,y) = (x+y)\ mod\ 2^k = \begin{cases} x+y, & \text{если } x+y < 2^k, \quad CF = 0 \\ x+y-2^k, & \text{если } x+y \geq 2^k, \quad CF = 1 \end{cases}$$

$$\text{разность}(x, y) = (x - y) \bmod 2^k = \begin{cases} x - y, & \text{если } x \geq y, \quad CF = 0 \\ (2^k + x) - y, & \text{если } x < y, \quad CF = 1 \end{cases}$$

Итак, сложение и вычитание беззнаковых целых чисел в ПК – это сложение и вычитание по модулю 2^k, где k – размер ячеек.

В ПК знаковые числа записываются в дополнительном коде: неотрицательное число записывается так же, как и беззнаковое число, а отрицательное число x представляется беззнаковым числом $2^k - |x|$, где k – количество разрядов в ячейке, отведенной под число.

$$\text{доп}(x) = \begin{cases} x, & \text{если } x \geq 0, \\ 2^k - |x|, & \text{если } x < 0, \end{cases}$$

Если целые числа со знаком представлены в дополнительном коде, то складывать и вычитать их можно по алгоритмам для беззнаковых чисел. Для этого дополнительные коды знаковых операндов рассматривают как числа без знака и в таком виде их складывают и вычитают, а полученный результат затем рассматривают как дополнительный код знакового ответа.

При сложении и вычитании как беззнаковых, так и знаковых чисел возможны особые случаи, когда настоящий (в математическом смысле) результат выходит за диапазон представимых чисел, и тогда результат искажается. Такое искажение результата фиксируется во флагах CF и OF (CF выставляется при переполнении по время операций над беззнаковыми числами - перенос, а OF – над знаковыми, если результат не помещается в представление числа – переполнение мантиссы). Распознать такую ошибку можно лишь последующим анализом этих флагов. Поскольку ПК заранее не знает над числами какого типа производятся операции, то при выполнении их ПК одновременно фиксирует во флагах CF и OF особенности операций для обоих классов чисел. Какие числа вычитаются/складываются, знает только автор программы.

При сложении и вычитании меняется также флаг нуля ZF и флаг знака SF. Флаг ZF получает значение 1, если результат оказался нулевым, и значение 0, если результат ненулевой; этот флаг представляет интерес при работе со знаковыми, так и с беззнаковыми числами. В флаг же SF заносится знаковый (самый левый) бит результата; этот флаг полезен при работе со знаковыми числами, так как он получает значение 1, если результат оказался отрицательным, и значение 0 иначе.

Примеры (ячейки размером с байт)

9 – 9 = 0 = 000000000b ==> ZF = 1, SF = 0

8 – 9 = -1 = 111111111b ==> ZF = 0, SF = 1

9 – 8 = 1 = 000000001b ==> ZF = 0, SF = 0

Сложение:	ADD op1, op2
Вычитание (substract):	SUB op1, op2

op1	*op2*	
r8 m8	i8, r8, m8 i8, r8	Сложение/вычитание байтов
r16 m16	i16, r16, m16 r16, m16	Сложение/вычитание слов

Команда ADD складывает операнды и записывает их сумму на место первого операнда: op1:=op1+op2. По команде SUB из первого операнда вычитается второй операнд и полученная разность записывается вместо первого операнда: op1:=op1-op2.

Пример:

ADD AH, 12 ; AH:=AH+12

SUB SI, Z ; SI:=SI-Z
ADD Z, -300 ; Z:=Z+(-300)

Увеличение на 1: INC op
Уменьшение на 1: DEC op
В этих командах допустимы следующие типы операнда: r8, m8, r16, m16
Примеры:
INC BL
DEC WORD PTR A
INC и DEC не меняют флага переноса CF.

Изменение знака (negative): NEG op
В этой команде допустимы следующие типы операнда: r8, m8, r16, m16. op:=-op
!Особый случай
MOV AH, -128 ; AH:=-128
NEG AH ; Не изменилось! AH:=-128, но OF = 1
При нулевом операнде CF = 0, при других CF = 1. Флаги SF и ZF меняются как обычно.

Сложение с учетом переноса (add with carry): ADC op1, op2
Вычитание с учетом заема (subtract with borrow): SBB op1, op2
Допустимые типы операндов – как в командах ADD и SUB.

Эти команды аналогичны командам обычного сложения и вычитания за одним исключением – в команде ADC к сумме операндов еще прибавляется значение флага переноса CF: op1:=op1+op2+CF, а в команде SBB из разности операндов еще вычитается значение этого флага: op1:=op1-op2-CF. Используются для сложения и вычитания чисел, размером в двойное слово.

Пример:

Пусть число X размещается в двух регистрах AX (старшие цифры) и BX (младшие), а число Y – в регистрах CX (старшие цифры) и DX (младшие), и если сумму двух чисел надо записать вместо числа X, тогда это делается так:

ADD BX, DX ; BX:=Xмл+Yмл, CF=перенос
SDC AX, CX ; AX:=Xст+Yст+CF

Для разности:
SUB BH, DX ; BX:=Xмл-Yмл, CF=перенос
SBB AX, CX ; AX:=Xст-Yст-CF

С помощью этих команд можно реализовать сложение и вычитание чисел любого размера.

Команды умножения и деления.

Команды умножения.

Умножение целых без знака (multiply): MUL op
Умножение целых со знаком (integer multiply): IMUL op

Умножение байтов: AX:=AL*op (op: r8, m8)
Умножение слов: (DX, AX):=AX*op (op: r16, m16)

Пример:

```
N DB 10
…
MOV AL, 2
MUL N               ; AX=2*10=20=0014h; AH=00h, AL=14h
MOV AL, 26
MUL N               ; AX=26*10=20=0104h; AH=01h, AL=04h
MOV AL, 8
MOV BX, -1
IMUL N              ; (DX,AX)=-8=0FFFFFFF8h; DX=0FFFFh, AL=0FFF8h
```

Команды умножения выдают результат в удвоенном формате. При этом:
CF=OF=1 – если произведение занимает двойной формат
CF=OF=0 - если произведению достаточен формат сомножителей

В процессоре 80186 была введена новая команда умножения с тремя операндами. Она допускает две эквивалентные формы записи:

```
MUL  op1, op2, op3
IMUL op1, op2, op3
```

а реализует следующее действие: op1 := op2 * op3

op1	*op2*	*op3*
r16	r16, m16	i16

Пример:

```
X DW ?
…
MUL  SI, BX, 2      ; SI:=BX*3
IMUL DX, X, -10     ; DX:=X*(-10)
```

Данная команда умножения предназначена для умножения только чисел размером в слово и только при условии, что произведение уменьшается в слово. При этом условии нет разницы между умножением чисел со знаком и чисел без знака, поэтому можно использовать любое из двух названий этой команды. Если указанное условие выполняется, тогда флаги CF и OF получают значение 0, но если результат превосходит размер слова, тогда левые, наиболее значимые, биты произведения теряются, в регистр op1 записываются последние 16 битов, а флаги CF и OF устанавливаются в 1.

Если op1=op2, то есть первый сомножитель берется из регистра, в который должен быть записан результат, тогда в записи команды этот регистр может быть указан только раз:

```
MUL  op1, op3
IMUL op1, op3
```

Пример:

```
MUL  DX, -15        ; DX:=DX*(-15)
```

Для использование данной команды в тексте программы (в любом месте, но до первой такой команды) должна быть помещена директива

```
.186
```

Пример:

```
.186                ; допускаются все команды процессора 80186
```

```
X DB ?
Y DW ?
…
MOV  AL, X
IMUL AL           ; AX:=X*X         (“обычное” умножение)
IMUL AX, 5  ; AX:=AX*5  (можно: MUL AX, 5 или IMUL AX, AX, 5)
IMUL BX, Y, -7    ; BX:=Y*(-7)
ADD  AX, BX
MOV  Y,  AX
```

Команды деления.

Деление целых без знака (divide): DIV op
Деление целых со знаком (integer divide): IDIV op

Деление слова на байт:
AH:=AX mod op, AL:=AX div op (op: r8, m8)

Деление двойного слова на слово:
DX:=(DX, AX) mod op, AX:=(DX, AX) div op (op: r16, m16)

Если через функцию trunc(x) обозначить отбрасывание дробной части вещественного числа x, тогда операции div и mod определяются следующим образом:

a div b = trunk(a/b)
a mod b = a-b*(a div b)

При выполнении команды деления возможно появление ошибки с названием «деление на 0 или переполнение». Она возникает в двух случаях:

- делитель равен 0 (op = 0)
- неполное частное не вмещается в отведенное ему место (регистр AL или AX); это, например, произойдет при делении 600 на 2;

```
MOV AX, 600
MOV BH, 2
DIV BH            ; 600 div 2 = 300, но 300 не вмещается в AL
```

При такой ошибке ПК прекращает выполнение программы.

Изменение размера числа.

Расширение байта до слова может быть реализовано следующим образом, приписав слева к числу нули:

MOV AH, 0 ; AL --> AX (без знака)

Расширение байта до слова может быть выполнено специальной командой:

Расширение байта до слова (convert byte to word): CBW

У этой команды местонахождение операнда и результата фиксировано: операнд всегда берется из AL, а результат всегда записывается в AX. Команда записывает число 00h или 0FFH в зависимости от знака числа из регистра AL.

$$AH := \begin{cases} 00h, & \text{при } AL \geq 0, \\ 0FFh, & \text{при } AL < 0, \end{cases}$$

Флаги эта команда не меняет.

Примеры:

```
MOV AL, 32  ; AL=20h
CBW                 ; AX=0020h  (число +32 как слово)
MOV AL, -32 ; AL=0F0h
CBW                 ; AX=0FFE0h (число -32 как слово)
```

Команда например используется при делении, когда требуется делимое (байт) расширить до слова:

```
; числа без знака
MOV AH, 0           ; AL -> AX (без знака)
DIV CH              ; AL:=AX div CH (AH:=AX mod CH)
; числа со знаком
CBW                 ; AL -> AX (со знаком)
IDIV CH             ; AL:=AX div CH (AH:=AX mod CH)
```

Для расширения слова, находящегося в регистре AX, до двойного слова, занимающего два регистра – DX и AX, при условии, что в DX находится старшая часть числа, а в AX – младшая AX→(DX, AX), используется команда CDW:

Расширение слова до двойного (convert byte to word): CWD

$$DX:=\begin{cases} 0000h, & \text{при } AX \geq 0, \\ 0FFFFh, & \text{при } AX < 0, \end{cases}$$

Пример 1.

Требуется переставить местами оба слова двойного слова X.

Решение.

```
MOV  AX, WORD PTR X              WP EQU WORD PTR
XCHG AX, WORD PTR X+2            MOV  AX,   WP X
MOV  WORD PTR X, AX              XCHG AX,   WP X+2
                                 MOV  WP X, AX
```

Пример 2.

Определить значения регистров AX и BX после выполнения команд:

```
W DW 1234h
MOV AX, W
MOV BH, BYTE PTR W
MOV BL, BYTE PTR W+1
```

Решение.

Ответ: AX=1234h, BX=3412h

Пример 3.

Пусть X – байтовая переменная, значение которой трактуется как знаковое число, а Y – переменная размером в слово. Вычислить Y=X*X*X при условии, что результат имеет размер слова.

Решение.

```
X DB ?
…
MOV AL, X
MUL AL              ; AX:=X*X (можно: MUL X)
MOV BX, AX          ; спасти AX
MOV AL, X
CBW                 ; AX:=X как слово
```

```
MUL BX              ; (DX, AX):=X*X*X (DX можно не учитывать)
MOV Y, AX
```

Пример 4.

```
N DB ?
D DB 3 DUP(?)
```

Рассматривая N как беззнаковое число (от 0 до 255), записать в массив D цифры (как символы) из десятичной записи этого числа; в байт D – левую цифру, в байт D+1 – среднюю цифру, в байт D+2 – правую цифру.

Решение.

Пусть a, b, c – десятичные цифры числа N: N=abc.

```
MOV BL,  10 ; делитель
MUL AL,  N
MOV AH,  0  ; AX:=N как слово
DIV BL              ; AH:=c, AL:=ab
ADD AH,  ‘0’ ; правая цифра как символ
MOV D+2, AH         ; запомнить c
MOV AH,  0  ; AX:=ab как слово
DIV BL              ; AH:=b, AL:=a (как числа)
ADD AX, ‘00’        ; обе цифры как символы (AH:=b+’0’, AL:=a+’0’)
MOV D+1, AH         ; запомнить b
MOV D,   AL ; запомнить a
```

<u>Лекция 5-7</u>

Переходы. Циклы.

Безусловный переход. Оператор SHORT

Переходы бываю условными и безусловными. Если переход делается только тогда, когда выполнено некоторое условие, то такой переход называется условным, а если он делается независимо от каких-либо условий, то это безусловный переход. В ПК команды перехода не меняют флаги.

Безусловный переход (jump): JMP op

Здесь операнд тем или иным способом указывает адрес перехода, т. е. адрес команды, которая должна быть выполнена следующей.

Прямой переход.

В данном случае в качестве op указывается метка той команды, на которую надо передать управление.

JMP <метка>

Пример:

```
  JMP L         ; следующей будет выполняться команда с меткой L
  …
L: MOV AX, 0
```

Меняет значение регистра IP! В ПК имеются две машинные команды прямого перехода, в одной из которых адрес перехода задается в виде байта (такая команда называется коротким переходом, адрес относительный), а другая – в виде слова (команда длинного перехода, абсолютный адрес)

Если мы заранее знаем, что переход вперед будет коротким, и если жалко терять байт на команде перехода, то мы должны предупредить ассемблер, что переход будет коротким. Для этого введен оператор SHORT:

```
JMP L           ; длинный переход (3 байта)
JMP SHORT L     ; короткий переход (2 байта)
  ...
L: ...
```

Если указали SHORT, но переход оказался длинным, то ассемблер зафиксирует ошибку.

Косвенный переход.

В этом случае в команде перехода указывается не сам адрес перехода, а то место, где находится этот адрес. Таким местом может быть регистр общего назначения или слово памяти:

JMP r16 или JMP m16

В этих командах берется содержимое указанного регистра или слова памяти, оно рассматривается как адрес некоторой команды программы и именно по этому адресу делается переход. При этом этот адрес рассматривается как «настоящий», а не отсчитанный от команды перехода.

Примеры ([x] – содержимое ячейки или регистра x)

```
A DW L
  ...
  JMP A            ; goto [A] = goto L
  ...
  MOV DX, A ; DX=L
  JMP DX           ; goto [DX] = goto L
  ...
L: ...
```

Косвенные переходы используются в тех случаях, когда адрес перехода становится известным только во время счета программы.

При переходах вперед имеем следующие случаи:

```
  JMP Z  ; goto Z        JMP Z  ; ошибка        JMP WORD PTR Z  ; goto Z
  ...                    ...                    ...
Z: ...                  Z DW L                 Z DW L
```

Команды сравнения и условного перехода.

Условный переход обычно реализуется в два шага: сначала сравниваются некоторые величины, в результате чего соответствующим образом формируются флаги (ZF, SF и т.д.).

Сравнение (compare): CMP op1, op2

Эта команда эквивалентна команде SUB op1, op2 за одним исключением: вычисленная разность op1-op2 никуда не записывается. Поэтому единственный и главный эффект от команды сравнения – это установка флагов, характеризующих полученную разность, или, что тоже самое, характеризующих сравниваемые величины op1 и op2.

Что же касается команд условного перехода, то их в ПК достаточно много, но они записываются единообразно:

Jxx <метка>

где операнд указывает метку той команды программы, на которую надо сделать переход в случае выполнения некоторого условия, а мнемокод начинается буквой J (от jump), за которой следует одна или несколько букв, в сокращенном виде описывающих это условие.

Все команды условного перехода можно разделить на три группы.

В первую группу входят команды, которые ставятся после команды сравнения. В их мнемокодах с помощью определенных букв описывается тот исход сравнения, при котором надо сделать переход. Эта такие буквы:

E – equal (равно)
N – not (не, отрицание)
G – greater (больше) – для чисел со знаком
L – less (меньше) – для чисел со знаком
A – above (выше, больше) – для чисел без знака
B – below (ниже, меньше) – для чисел без знака

Мнемокод	*Содержательное условие для перехода после CMP op1, op2*	*Состояние флагов для перехода*
	для любых чисел:	
JE	op1 == op2	ZF=1
JNE	op1 != op2	ZF=0
	для чисел со знаком:	
JL/JNGE	op1 < op2	SF<>OF
JLE/JNG	op1 =< op2	SF<>OF или ZF=1
JG/JNLE	op1 > op2	SF=OF, ZF=0
JGE/JNL	op1 >= op2	SF=OF
	для чисел без знака:	
JB/JNAE	op1 < op2	CF=1
JBE/JNA	op1 =< op2	CF=1 или ZF=1
JA/JNBE	op1 > op2	CF=0, ZF=0
JAE/JNB	op1 >= op2	CF=0

Пример. Пусть X, Y, Z – переменные размером в слово. Требуется записать в Z максимальное из чисел X и Y.

```
; числа со знаком
  MOV AX, X
  CMP AX, Y
  JGE M
  MOV AX, Y
M: MOV Z, AX
```

```
; числа без знака
  MOV AX, X
  CMP AX, Y
  JAE M
  MOV AX, Y
M: MOV Z, AX
```

Во вторую группу команд условного перехода входят те, которые ставятся после команд, отличных от команды сравнения, и которые реагируют на то или иное значение какого-нибудь определенного флага. В мнемокодах этих команд указывается первая буква проверяемого флага, если переход должен выполнен при значении 1 у флага, либо эта буква указывается с буквой N (not), если переход надо сделать при нулевом значении флага.

Мнемокод	***Условие перехода***
JZ	ZF=1
JS	SF=1
JC	CF=1
JO	OF=1

Мнемокод	***Условие перехода***
JNZ	ZF=0
JNS	SF=0
JNC	CF=0
JNO	OF=0

Пример. Пусть A, B, C – беззнаковые байтовые переменные. Требуется вычислить C=A*A+B, но если ответ превосходит размер байта, тогда надо передать управление на метку ERROR.

```
MOV AL, A
MUL AL
JC ERROR          ; A*A > 255(CF=1) --> ERROR
ADD AL, B
JC ERROR          ; перенос (CF=1) --> ERROR
MOV C, AL
```

В третью группу входит только одна команда условного перехода, проверяющая не флаги, а значение регистра CX:

JCXZ <метка>

Действие команды JCXZ (jump if CX is zero) можно описать так:
if CX = 0 then goto <метка>

Все команды условного перехода осуществляют только короткий переход (на 30-40 команд).

Длинный переход выполняется примерно так:
if AX=BX then goto M
следует реализовывать:

```
if AX<>BX then goto L;     // короткий переход
goto M;                    // длинный переход
L: …
```

На ассемблере:

```
    CMP AX, BX
    JNE L
    JMP M
L:...
```

Команды управления циклом.

if (X>0) then
  S1
else
  S2
	while (X > 0) {
	  S
	};
		do {
		} while (X > 0);

CMP X, 0
  JLE L2
    S1
  JMP FIN
L2:
    S2
FIN:
	BEG: CMP X, 0
	    JLE FIN
	      S
	    JMP BEG
	FIN:
		BEG: S
		    CMP X, 0
		    JG BEG

Команда LOOP

Пусть некоторую группу команд (тело цикла) надо повторить N раз (N>0). Тогда этот цикл можно реализовать по такой схеме:

```
    MOV CX, N       ; CX – счетчик цикла (число повторений)
L:  ...             ;
    ...             ; тело цикла
    ...             ;
    DEC CX          ; CX:=CX-1
    CMP CX, 0       ; CX=0?
    JNE L           ; CX<>0 goto L
```

Последние три команды можно заменить командой LOOP (реализует короткий переход):

```
    MOV CX, N       ; N > 0
L:  ...             ;
    ...             ; тело цикла
    ...             ;
    LOOP L          ;
```

Действие этой команды описывается так:

CX:=CX-1: If (CX<>0) then goto <метка>

Если возможен вариант, что число повторений может быть нулевым, то при CX=0 надо делать обход цикла:

```
    MOV CX, N       ; N >= 0
    JCXZ L1         ; CX = 0 -> L1
L:  ...             ;
    ...             ; тело цикла
    ...             ;
    LOOP L          ;
L1: ...
```

Пример. Вычислить AX:=N! (N<8)

```
    MOV AX, 1       ; AX:=1
    MOV CL, N
    MOV CH, 0       ; CX:=N как слово (счетчик цикла)
    JCXZ F1         ; При N=0 обойти цикл
    MOV SI, 1       ; i:=1
F:  MUL SI          ; (DX, AX):=AX*I (DX=0)
    INC SI          ; i:=i+1
    LOOP F
F1: ...
```

или еще проще, но не нагляднее:

```
    MOV AX, 1
    MOV CL, N
    MOV CH, 0       ; CX – и счетчик и параметр цикла
    JCXZ F1
F:  MUL CX
    LOOP F
F1: ...
```

Команды LOOPE/LOOPZ и LOOPNE/LOOPNZ

Эти команды похожи на команду LOOP, то есть заставляют повториться столько раз, сколько указано в регистре CX, однако они допускают и досрочный выход из цикла.

Цикл по счетчику и пока равно (пока ноль): LOOPE/LOOPZ <метка>

Действие этой команды описывается так:
CX:=CX-1: If (CX<>0) and (ZF=1) then goto <метка>

Чаще всего команда используется для поиска первого элемента некоторой последовательности, отличного от заданной величины.

Цикл по счетчику и пока не равно (пока не ноль): LOOPE/LOOPZ <метка>
Действие этой команды описывается так:
CX:=CX-1: If (CX<>0) and (ZF=0) then goto <метка>

Команда используется обычно для поиска первого элемента некоторой последовательности, имеющего заданную величины.

Домашнее задание.
1. *Записать в регистр BL наименьшее число из отрезка [2, K], на которое не делится число N, или записать 0, если такого числа нет.*
2. *Пусть A, B, C, M – переменные размером в слово. Требуется записать в M максимальное из чисел A, B, C*

<u>Лекция 8-10</u>

Строковые команды. Префиксы повторения

Строки - столь важный тип данных, что во многие ЭВМ вводят специальные команды, упрощающие обработку строк. Особенность этих команд, называемых строковыми, в том, что одной или парой таких команд можно выполнить некоторую операцию сразу над всей строкой. Есть такие команды и в ПК. У строковых команд ПК много общего, поэтому, чтобы не повторяться, мы сначала подробно рассмотрим одну из них, а затем коротко опишем остальные.

Команда сравнения строк

Прежде всего отметим, что в строковых командах под "строкой" понимается не только последовательность байтов (символов), но и последовательность слов. В связи с этим каждая строковая операция представлена в ПК двумя командами: одна из них предназначена для обработки строк из байтов, а другая - для обработки строк из слов. В ЯА мнемокоды этих команд различаются тем, что в первом случае указывается буква B (byte), а в другом - буква W (word). Например, в командах сравнения строк (CMPS, compare strings) используются следующие мнемокоды:

CMPSB - сравнение строк из байтов
CMPSW - сравнение строк из слов.

В целом действия этой пары команд совпадают, поэтому обычно про них говорят как про одну команду - команду CMPS, и только если надо, уточняют, какой именно вариант ее имеют в виду.

Команда CMPS записывается без операндов, ее действие можно описать так:

[DS:SI]=[ES:DI]? ; SI:=SI+d; DI:=DI+d,

где величина d определяется согласно следующей таблице (DF - флаг направления, direction flag):

	DF=0	*DF=1*
CMPSB	+1	-1
CMPSW	+2	-2

Прокомментируем действия команды CMPS.

У команды два операнда, но их местонахождение заранее известно, поэтому они явно и не указываются. CMPS принято называть командой сравнением строк, но на самом деле она сравнивает только пару элементов - элемент из одной строки с элементом из другой строки. При этом абсолютный адрес элемента из первой строки должен задаваться регистрами DS и SI, а абсолютный адрес элемента из второй строки - регистрами ES и DI.

Сравниваемые строки могут находиться в памяти далеко друг от друга, в разных сегментах памяти. Поэтому в общем случае строки нельзя сегментировать по одному сегментному регистру. В связи с этим в ПК договорились о том, что одна строка должна сегментироваться по регистру DS, т. е. DS должен указывать на начало сегмента памяти, в котором находится эта строка, а другая строка должна сегментироваться по регистру ES. Смещения же сравниваемых элементов (их адреса, отсчитанные от начала тех сегментов памяти, где расположены строки) договорились указывать в регистре SI (для элемента первой строки) и регистре DI (для элемента второй строки). Подчеркнем особо: в регистрах SI и DI указываются адреса элементов строк, а не их индексы, не их номера внутри строк.

Основное действие команды CMPS заключается в сравнении элемента одной строки с элементом другой строки. При этом команда CMPSB сравнивает байты, а команда CMPSW - слова. Это сравнение выполняется так же, как и в команде обычного сравнения CMP, т. е. путем вычитания операндов без записи куда-либо полученной разности. Главное здесь - формирование флагов, которые отражают результат сравнения и которые затем можно проверить командами условного перехода.

Но на этом действие команды не заканчивается. Она еще меняет значения регистров SI и DI и меняет так, чтобы в них оказались адреса соседних элементов строк. Но каких соседних? Это зависит от текущего значения флага направления DF: при DF=0 значения регистров увеличиваются, т. е. происходит переход вперед - к следующим элементам, а при DF=1 значения регистров уменьшаются, т. е. происходит переход назад - к предыдущим элементам. Разница же между командами CMPSB и CMPSW проявляется в том, что команда CMPSB меняет эти регистры на 1, а команда CMPSW - на 2, т.е. эти команды учитывают размеры элементов строк.

Как видно, действие команды зависит от значения флага направления DF. Сам по себе этот флаг не меняется, менять его должен автор программы. Сделать это можно с помощью следующих команд:

Очистка флага DF (clear DF): CLD
Установка флага DF (set DF): STD

По команде CLD флаг направления обнуляется (DF:=0), а по команде STD флагу присваивается 1 (DF:=1). Установленное этими командами значение флага сохраняется до тех пор, пока не будет снова выполнена одна из этих команд. Отметим также, что в самом начале выполнения программы значение флага DF может быть любым.

Итак, команда CMPS сравнивает пару элементов двух строк и автоматически настраивается на соседние элементы строк, обеспечивая продвижение по строкам в определенном направлении - вперед (от начала к концу) при DF=0 или назад при DF=0.

Легко сообразить, что осталось только зациклить эту команду, чтобы можно было сравнить строки целиком.

Пусть N - это некоторая константа с положительным значением и пусть строка S1 из N бантов расположена в сегменте памяти, на начало которого уже установлен сегментный регистр DS, а строка S2 из того же числа байтов находится в сегменте, на начало которого уже установлен регистр ES. Тогда проверить на равенство эти две строки можно с помощью следующих команд (слева строки просматриваются вперед, справа - назад):

```
; просмотр вперед                        ; просмотр назад
   CLD        ;DF=0 (вперед)                STD            ;DF=1 (назад)
   LEA SI,S1  ;DS:SI=начало S1              LEA SI,S1+N-1  ;DS:SI=конец S1
   LEA DI,S2  ;ES:DI=начало S2              LEA DI,S2+N-1  ;ES:DI=конец S2
   MOV CX,N   ;CX=длина строк               MOV CX,N       ;CX=длина строк
L: CMPSB      ;сравнить пару эл-тов      L: CMPSB      ;сравнить пару эл-тов
   JNE NOEQ   ;S1<>S2 --> NOEQ              JNE NOEQ       ;S1<>S2 --> NOEQ
   LOOP L     ;к следующей паре             LOOP L         ;к предыдущей паре
EQ: ...       ;S1=S2                     EQ: ...           ;S1=S2
```

Как видно, в этих циклах нам не пришлось самим менять значения регистров SI и DI, это делает строковая команда. Собственно в этом и проявляется достоинство строковых команд, этим они и упрощают реализацию операций над строками. Однако создатели ПК на этом не успокоились и предоставили средство, упрощающее зацикливание строковых команд. Это префиксы повторения.

Префиксы повторения

В ПК имеется две команды без операндов, которые называются префиксами повторения. В ЯА каждая из них имеет несколько названий-синонимов:

1-й префикс: REPE (repeat: if equal; повторять, пока равно),
REPZ (повторять, пока ноль),
REP (повторять)

2-й префикс: REPNE (repeat if not equal; повторять пока не равно),
REPNZ (повторять, пока не ноль)

(Сравнение x=y можно представить как сравнение x-y=0, что и объясняет эквивалентность фраз "пока равно" и "пока ноль". О названии же просто REP будет сказано позже).

Префиксы повторения ставятся перед строковыми командами, причем в ЯА такой префикс обязательно должен записываться в одной строчке со строковой командой; например:

REPE CMPSB

Если поставить префикс повторения перед нестроковой командой, то он никак не будет сказываться, проработает как "пустая" команда. Действие префикса повторения состоит в том, что он заставляет многократно повторяться следующую за ним строковую команду. При этом данная пара команд выполняется значительно быстрее, чем цикл с этой строковой командой, который мы организуем сами.

Точный смысл пары команд

REPE <строковая команда>

следующий (ZF - флаг нуля):

```
L: if CX=0 then goto Ll;
   CX:=CX-1;
   <строковая команда>
   if ZF=l then goto L;
Ll:
```

Легко заметить, что префикс заставляет повторяться строковую команду такое число раз, которое указано в регистре CX, т. е. префикс использует этот регистр как счетчик цикла. Число повторений (оно всегда рассматривается как целое без знака) должно быть записано в регистр CX, конечно, до выполнения этой пары команд. Если это 0, то строковая команда не выполняется ни разу. Иначе префикс заставляет выполняться строковую команду и вычитает 1 из CX. И так до тех пор, пока в CX не окажется 0.

Однако это не единственная причина прекращения цикла. Как видно, после каждого выполнения строковой команды проверяется флаг нуля ZF. Если он равен 1, то цикл продолжается, но если после очередного выполнения строковой команды флаг ZF получит значение 0, то цикл тут же прекращается. А что означают значения 1 и 0 у флага ZF? Например, если строковая команда - это CMPS, то ZF=1 означает равенство сравниваемых элементов, а ZF=0 - неравенство.

Таким образом, цикл продолжается, пока сравниваемые элементы равны, и прекращается, как только нашлась пара неравных элементов.

В целом, действие пары команд REPE CMPS можно описать так: повторяй сравнение элементов строк, пока они равны, но не более CX раз.

Итак, есть две причины выхода из цикла - либо строки просмотрены полностью и все их элементы оказались равными (тогда CX=0 и ZF=1), либо нашлась пара неравных элементов (тогда ZF=0, а CX может быть любым). По какой именно причине произошел выход, надо устанавливать после выхода из цикла анализом флага ZF с помощью команд условного перехода. Например, приведенные выше команды сравнения строк S1 и S2 (при просмотре вперед), можно переписать так:

```
        CLD                 ; DF=0 (вперед)
        LEA SI,SI           ; DS:SI=начало Sl
        LEA DI,S2           ; ES:DI=начало S2
        MOV CX,N            ; CX=длина строк
        REPE CMPSB          ; сравнивать, пока элементы равны (пока ZF = 1)
        JE EQ         ; ZF=1 (S1=S2) ---> EQ
NOEQ:...                    ; ZF=0 (S1<>S2)
```

Теперь рассмотрим некоторые детали, которые важно знать при использовании префиксов повторения.

Во-первых, префикс повторения можно использовать не только при проверке строк на равно/не равно, но и при проверке на больше/меньше. В самом деле, если мы вышли из цикла при ZF=1, то значит, все элементы строк равны, но если оказалось ZF=0, то какая-то пара элементов строк различается. Но как различается? Напомним, что команда CMPS, как и команда CMP, меняет не только флаг ZF, но и другие флаги; например, в нашем случае будет CF=1. Поэтому проверкой других флагов и можно определить, какой из неравных элементов больше, а какой - меньше.

Если дописать в конец нашей группы команд сравнения строк команды

```
NOEQ:  JA GREATER       ; S1>S2 ->  GREATER
```

```
LESS: ...          ; S1<S2
```

то получим проверку всех трех возможных исходов сравнения: при S1=S2 будет переход на метку EQ, при S1>S2 - на метку GREATER, а при S1<S2 - на метку LESS.

Во-вторых, по выходу из цикла в регистрах SI и DI оказываются адреса не тех элементов, на которых прекратилось сравнение, а следующих за ними. В нашем примере неравными оказались третьи элементы, но эти регистры будут показывать на четвертые элементы. Это объясняется тем, что команда CMPS после сравнения очередной пары элементов всегда меняет регистры SI и DI, равны эти элементы или нет.

В-третьих, после выхода из цикла в регистре CX всегда находится число элементов строк, оставшихся нерассмотренными, которые не сравнивались. Это очевидно, если вспомнить, что в начале в CX было число всех элементов строк и что после каждого сравнения из CX вычитается 1.

Нетрудно заметить, что после выхода из цикла регистры SI, DI и CX имеют такие значения, которые позволяют продолжить сравнение оставшихся частей строк (после пары неравных элементов). Этим, например, можно воспользоваться для подсчета числа неравных пар в строках:

```
    MOV AX,.O       ;AX - число неравных пар
    CLD
    LEA SI,SI
    LEA DI,S2
    MOV CX,N
COMP: REPE CMPSB
    JE FIN      ;переход на FIN, если вышли из цикла при равной
                ;паре, т. е. дошли до конца строк
    INC AX          ;учесть очередную неравную пару
    CMP CX,0        ;продолжить сравнение "хвостов" строк,
    JNE COMP        ;если они не пусты
 FIN: ...
```

И, наконец, последнее замечание. Если до цикла значение регистра CX было нулевым, то значение флага ZF после цикла будет таким же, как и до цикла, поэтому по ZF нельзя определять причину выхода из "пустого" цикла. Это надо учитывать при сравнении пустых строк. Теперь рассмотрим другой префикс повторения. Здесь все аналогично, но с заменой значения флага ZF на противоположное. А именно пара команд

```
        REPNE <строковая  команда>
```

выполняется следующим образом:

```
L: if CX=0 then goto  Ll;
  CX:=CX-1;
  <строковая команда>
  if ZF=0 then goto  L;
Ll:
```

Смысл префикса REPNE перед командой CMPS можно сформулировать так: повторяй сравнение элементов строк, пока они не равны, но не более CX раз. Значит, пока сравниваемые элементы строк различны, сравнение продолжается, но как только будет найдена пара равных элементов, цикл прекращается. Пара команд REPNE CMPS используется, когда надо найти первую (от начала или конца) пару равных элементов строк.

Другие строковые команды

Теперь рассмотрим другие строковые команды, но уже бегло, т. к. они во многом аналогичны команде сравнения строк.

Сканирование строки (SCAS, scan string): SCASB
SCASW

По команде SCASB содержимое регистра AL сравнивается с байтом памяти, абсолютный адрес которого указывает пара регистров ES:DI, после чего регистр di автоматически устанавливается на соседний байт памяти:

AL=[ES:DI]? ; DI:=DI±1

(сравнение происходит так же, как в команде CMP AL, ES:[DI]; плюс берется при DF=0, а минус - при DF=1). Команда же SCASW сравнивает слова - из регистра AX и ячейки памяти, абсолютный адрес которого определяется парой ES:DI, после чего регистр DI также автоматически устанавливается на соседнее слово памяти:

AX=[ES:DI]? ; DI:=DI±2

Команда SCAS (в любом варианте) используется для поиска в строке элемента, равного заданному (в AL или AX) или отличного от заданного - в зависимости от того, какой префикс повторения поставлен перед ней:

REPNE SCASB - найти в строке первый элемент, равный AL
("повторяй сравнение, пока элементы не равны AL")
REPE SCASB - найти в строке первый элемент, отличный от AL
("повторяй сравнение, пока элементы равны AL")

Используется например, когда требуется заменить первое вхождение определенного символа в строке.

Пересылка строки (MOVS, move string): MOVSB
MOVSW

Команда MOVSB пересылает байт, а команда MOVSW - слово, абсолютный адрес которого задается парой регистров DS:SI, в ячейку, абсолютный адрес которой задается парой ES:DI, после чего значения регистров SI и DI автоматически меняются так, чтобы они указывали на соседние элементы (правила изменения этих регистров такие же, как и в команде CMPS):

[DS:SI] => [ES:DI]; SI:=SI+d; DI:=DI+d

Флаги эти команды не меняют.

Команда MOVS (в любом варианте) пересылает только один элемент строки, но зато сразу настраивается на работу с соседним элементом. Поэтому осталось лишь зациклить эту команду, чтобы можно было переслать всю строку из одного места памяти в другое. Для такого зацикливания можно воспользоваться любым префиксом повторения, т. к. команда MOVS не меняет флаги и потому выход из цикла возможен только по одной причине - по CX=0, т. е. когда будут переписаны все элементы строки. Обычно перед командой MOVS указывается префикс с названием REP, т. е. просто "повторяй" без добавки "пока равно" или "пока не равно", т. к. они здесь малоосмысленны:

```
REP MOVSB  ;повторяй пересылку (байтов) CX раз
```

Основное назначение команды MOVS - быстрая перепись содержимого одной области памяти в другую. Например, если в сегменте данных имеются массивы

```
X DW 100 DUP(?)
Y DW 100 DUP(?)
```

и требуется выполнить присваивание X:=Y, то сделать это можно так:

```
CLD           ;просмотр вперед
LEA SI,Y      ;DS:SI=ачало Y ("откуда")
PUSH DS
POP ES        ;установка ES на сегмент данных
LEA DI,X      ;ES:DI=начало X ("куда")
MOV CX,100    ;сколько слов переписывать
REP MOVSW
```

Сохранение строки (STOS, store string):   STOSB
                                            STOSW

По команде STOSB в байт памяти, абсолютный адрес которого задается парой регистров ES:DI, записывается содержимое регистра AL, после чего значение регистра DI меняется на +1 при DF=0 или на -1 при DF=1. Команда STOSW записывает содержимое регистра AX в слово памяти, абсолютный адрес которого задается регистрами ES:DI, после чего меняет значение регистра DI на ±2. Флаги эти команды не меняют.

Перед командой STOS имеет смысл указывать только префикс REP, и тогда пару команд можно использовать для записи во все ячейки какой-то области памяти одной и той же величины - той, что указана в регистре AL или AX. Например, заполнить пробелами всю 40-символьную строку S из сегмента данных можно так:

```
MOV AL,' '    ;символ для записи
CLD           ; просмотр вперед
PUSH DS
POP ES
LEA DI,S      ;ES:DI=Ha4ano S
MOV CX,40     ;число заполняемых байтов
REP STOSB
```

Загрузка строки (LODS, load string):   LODSB
                                         LODSW

Команда LODSB (LODSW) записывает в регистр AL (AX) содержимое байта (слова) памяти, абсолютный адрес которого задается регистрами DS:SI, после чего меняет значение регистра SI на ±1 (на ±2). Флаги не меняются. Указывать префикс повторения перед командой LODS бессмысленно. Обычно эта команда используется вместе с командой STOS для переписи строк, когда между считыванием и записью элементов строк над ними должна быть выполнена какая-то дополнительная операция. Например, если в сегменте данных имеются байтовые массивы X и Y из 101 знакового числа в каждом, тогда переписать все числа из X в Y с изменением их знака можно так:

```
        CLD        ;просмотр вперед
        LEA SI,X     ;DS:SI - "откуда" (для команды LODS)
        PUSH DS
        POP ES
        LEA DI,Y     ;ES:DI - "куда" (для команды STOS)
        MOV CX,101   ;сколько переписывать
L:      LODSB        ;AL <- очередное число из X; SI:=SI+1
        NEG AL       ;изменение его знака
        STOSB        ;AL -> очередной элемент в Y; DI:=DI+1
        LOOP L
```

<u>Лекция 11</u>

Команды загрузки адресных пар в регистры

Использование строковых команд и префиксов повторения позволяет существенно ускорить обработку строк, однако перед этими командами приходится вписывать достаточно много "установочных" команд (установить флаг направления, записать в регистр CX число повторений и т.д.). В определенной мере сократить число таких команд позволяют следующие две команды ПК, которые загружают в регистры адресные пары (указатели) и с помощью которых можно Установить пары регистры DS:SI и ES:DI на обрабатываемые строки.

Загрузка указателя в DS (load pointer using DS): LDS rl6, m32

В качестве второго операнда (m32) должен быть указан адрес двойного слова памяти, в котором находится пара seg:ofs, задающая абсолютный адрес некоторой ячейки памяти (из-за "перевернутого" представления двойных слов в памяти ПК часть ofs должна находиться в первом слове этого двойного слова, часть seg , во втором слове), а в качестве первого операнда - название любого регистра общего назначения. Команда записывает в этот регистр смещение ofs, а в регистр DS - номер сегмента seg:

rl6:=[m32], DS:=[m32+2]

Флаги команда не меняет.

Загрузка указателя в ES (load pointer using ES): LES rl6,m32

Эта команда полностью аналогична команде LDS, только вместо регистра DS здесь используется регистр ES.

Пример:

```
DATA1 SEGMENT
  SI DB 400 DUP(?)
 AS2 DD S2           ;AS2 = DATA2:S2
DATA1 ENDS
DATA2 SEGMENT
 S2 DB 400 DUP(?)
DATA2 ENDS
CODE SEGMENT
   ASSUME CS:CODE, DS:DATA1
       ...
```

```
; пусть в этот момент DS=DATA1
        CLD
        LEA SI,SI                ;DS:SI=Начало SI
        LES DI,AS2          ;ES:DI=Начало S2
        MOV CX,400
        REP MOVSB                ;копирование строки SI в строку S2
```

Строки переменной длины

В зависимости от того, меняются в процессе выполнения программы длины строк или нет, строки принято делить на строки переменной длины и строки фиксированной длины. Строки фиксированной длины - это обычные массивы. Нас будут интересовать строки переменной длины, причем только символьные строки (строки из байтов), поскольку обычно именно их имеют в виду, когда говорят про строки.

При работе со строкой переменной длины сразу возникает вопрос: сколько места отводить под нее? Ведь при составлении программы мы обязаны описать строку, явно указав, сколько места мы отводим под нее, но в это время длина строки еще неизвестна.

Возможный выход здесь - заранее узнать максимально возможную длину строки и отвести место в памяти под строку по этому максимуму. Например, если известно, что в любой момент строка S будет содержать не более 200 символов, тогда надо отвести под нее 200 байтов:

```
S DB 200 DUP(?)  ;длина(S)<=200
```

При этом символы, входящие в данный момент в строку, всегда будем располагать в начале выделенного участка памяти. Отметим, что если заранее неизвестна и максимальная длина, тогда надо выкручиваться как-то по-другому, например, надо воспользоваться списками.

Поскольку длина строки может меняться, то надо как-то узнавать текущую длину строки. Решить эту проблему можно двояко.

Первый вариант - выбираем некоторый символ (скажем, с кодом 0), о котором заранее известно, что он не будет входить в строку, и договариваемся размещать его всегда за последним символом строки, т. е. используем его как признак конца строки:

S	s1	s2	...	sn	0	

При этом содержимое байтов за этим спецсимволом считается не относящимся к строке. Если длина строки меняется, то спецсимвол сдвигается.

Этот способ представления строк переменной длины используется достаточно часто на практике (например, в языке Си). Но у него есть крупный недостаток: не просмотрев строку до конца, нельзя определить ее длину, а это может привести к лишней работе. Например, если надо проверить на равенство строку из 100 символов и строку из 50 символов, то ясно заранее, что эти строки не равны, однако при данном представлении строк мы можем узнать об этом, только просмотрев первые 50 символов.

Текущая длина строки легко узнается при другом представлении строк переменной длины, в котором начальная ячейка строки отводится для хранения текущей длины строки, а сами символы строки размещаются вслед за этой ячейкой:

	0	1	2		n	
S	n	s1	s2	...	sn	

Содержимое байтов, следующих за n-ым символом, считается не относящимся к текущему значению строки.

А сколько места отводить под длину строки (под n)? Это зависит от максимальной длины строки. Если эта длина больше 255, тогда, как минимум, надо отмсти слово, а если она не превосходит 255, то достаточно и одного байта. Для определенности договоримся, что у нас строки будут содержать не более 255 символов, поэтому для хранения длины строк будем выделять один байт. При этом будем считать, что этот байт имеет индекс 0, тогда i-й символ строки окажется в байте с индексом i, т. е. адрес(S[i])=S+i.

Поскольку мы отводим место не только для символов строки, но для ее текущей длины, то этот дополнительный байт надо учитывать при описании строки. Например, если заранее известно, что в строке не будет более 200 символов, то под нее надо отвести 201 байт:

```
S DB 201 DUP(?)  ;длина(S)<=200
```

Массивы. Структуры.

В ассемблера массивы описываются по директивам определения данных с использованием конструкции повторения DUP.

Пусть имеется массив X из 30 элементов-слов:

```
X DW 30 DUP(?)
```

Индексация в массивах может определяться произвольным образом, однако обычно полагают, что элементы массива нумеруются с 0, то есть

```
X DW 30 DUP(?) ; X[0..29]
```

Причем справедливо следующее соотношение:

адрес(X[i]) = X + (type X) * i

Для многомерных массивов ситуация аналогична. Пусть, к примеру, имеется двумерный массив (матрица) A, в котором N строк и M столбцов (N и M - константы) и все элементы - двойные слова:

```
A DD N DUP(M DUP (?))
```

Здесь мы предполагаем, что элементы матрицы размещаются в памяти по строкам: первые M ячеек (двойных слов) занимают элементы первой строки матрицы, следующие M ячеек - элементы второй строки и т. д. Конечно, элементы матрицы можно размещать и по столбцам, но традиционно принято построчное размещение.

При этом предположении зависимость адреса элемента матрицы от индексов элемента выглядит так:

адрес(A[i, j]) = A + M * (type A) * i + (type A) * j

<u>Лекция 12-13</u>

Модификация адресов

В общем случае в команде вместе с адресом может быть указан в квадратных скобках некоторый регистр, например: MOV CX, A[BX]. Тогда команда будет работать не с указанным в ней адресом A, а с так называемым исполнительным (другое название - эффективным) адресом Аисп, который вычисляется по следующей формуле:

Аисп = (A + [BX]) mod 2^{16}

где [BX] обозначает содержимое регистра BX. Другими словами, прежде чем выполнить команду, центральный процессор прибавит к адресу A, указанному в команде, текущее содержимое регистра BX, получит некоторый новый адрес и именно из ячейки с этим адресом возьмет второй операнд. (Сама команда при этом не меняется, суммирование происходит внутри центрального процессора.) Если в результате сложения получилась слишком большая сумма, то от нее берутся только последние 16 битов, на что и указывает операция mod в приведенной формуле. (Отметим, что если в команде рядом с адресом не указан регистр, то исполнительный адрес считается равным адресу из команды.)

Замена адреса из команды на исполнительный адрес называется модификацией адреса, а регистр, участвующий в модификации, принято называть регистром-модификатором или просто модификатором. Отметим при этом, что в ПК в качестве модификатора можно использовать не любой регистр, а только один из следующих четырех: BX, BP, SI или DI.

Одним из случаев, где полезна модификация адресов, является индексирование, используемое для реализации переменных с индексом.

Индексирование

Пусть имеется массив:

X DW 100 DUP(?) ; X[0..99]

и требуется записать в регистр AX сумму его элементов.

Для нахождения суммы надо сначала в AX записать 0, а затем в цикле выполнять операцию AX:=AX+X[i] при i от 0 до 99. Поскольку адрес элемента X[i] равен X+2*i, то команда, соответствующая этой операции, должна быть следующей:

ADD AX, X+2*i

Но такая команда запрещена правилами и машинного языка: в любой команде все ее части, в том числе и адрес, должны быть фиксированными, не должны меняться. У нас же адрес меняется вместе с изменением индекса i.

Итак, мы столкнулись со следующей проблемой: по алгоритму наша команда должна работать с разными адресами (как говорят, должна работать с переменным адресом), а правила машинного языка допускают только фиксированный адрес. Именно для устранения этого противоречия и была введена в вычислительные машины модификация адресов, с помощью которой эта проблема решается следующим образом.

Разобьем переменный адрес X+2*i на два слагаемых - на постоянное слагаемое X, которое не зависит от индекса i, и на переменное слагаемое 2*i, зависящее от индекса. Постоянное слагаемое записываем в саму команду, а переменное слагаемое заносим в какой-нибудь регистр-модификатор (скажем, в SI) и название этого регистра также записываем в команду в качестве модификатора:

ADD AX, X[SI]

Поскольку регистр-модификатор один и тот же, то такая команда имеет фиксированный вид, т. е. удовлетворяет правилам машинного языка. С другой стороны, команда работает с исполнительным адресом, а он получается сложением адреса X из команды с содержимым (2*i) регистра SI, которое может меняться. Поэтому, меняя значение SI, мы заставим неменяющуюся команду работать с разными адресами. Тем самым удовлетворяются требования как машинного языка, так и алгоритма. Единственное, что осталось сделать, - это правильно менять содержимое регистра SI. Но это уже делается просто: вначале в SI надо заслать 0, а затем увеличивать его значение с шагом 2; в результате наша команда будет работать с адресами X, X+2, X+4,…, X+198.

Поскольку в данном случае регистр-модификатор используется для хранения индекса (точнее - выражения, зависящего от индекса), то такой регистр называют индексным регистром, а описанный способ получения адреса переменной с индексом - индексированием.

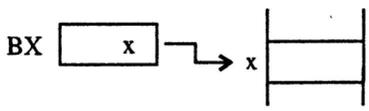
С учетом всего сказанного фрагмент программы нахождения суммы элементов массива X выглядит так:

```
        MOV AX, 0           ; начальное значение суммы
        MOV CX, 100         ; счетчик цикла
        MOV SI, 0           ; начальное значение (удвоенного индекса)
L:      ADD AX, X[SI]       ; AX:=AX+X[i]
        ADD SI, 2           ; следующий индекс
        LOOP L              ; цикл 100 раз
```

Косвенные ссылки

Рассмотрим еще один случай применения модификации адресов.

Пусть нам надо решить следующую задачу: имеется некоторая ячейка размером слово, адрес которой нам не известен, но известно, что этот адрес находится регистре BX, и надо записать, скажем, число 300 в эту ячейку:



Если бы мы заранее знали адрес (x) этой ячейки, то наша задача решалась бы командой MOV x, 300. Но в момент составления программы мы не знаем этот адрес, а потому и не можем указать его в команде. Что делать? Вспомним, что команды работают с исполнительными адресами, поэтому нам надо взять такой адрес и такой модификатор, чтобы в сумме они давали этот заранее не известный нам адрес. Легко сообразить, что в нашем случае надо взять нулевой адрес и регистр BX, поскольку тогда Аисп=0+[BX]=0+x=x. Поэтому наша задача решается командой

```
        MOV [BX],300
```

Особенность используемого здесь способа модификации адреса заключается в том, что мы, как и прежде, представляем адрес в виде суммы двух слагаемых, одно из которых записываем в команду, а другое - в регистр, но если ранее у нас оба слагаемых были ненулевыми, то теперь одно слагаемое, которое мы помещаем в команду, нулевое и потому весь адрес "упрятан" в регистре. Получается, что в команде мы указывается лишь место (регистр), где находится адрес. Такой способ задания адреса через промежуточное звено называют косвенной ссылкой или косвенной адресацией.

Отметим попутно, что при косвенной ссылке обычно приходится уточнять размер ячейки, на которую она указывает. Если, к примеру, в ячейку, адрес которой находится в регистре BX, надо записать число 0, тогда использовать для этого команду MOV [BX], 0 нельзя, т. к. в ней непонятны размеры операндов: 0 может быть как байтом, так и словом, да и адрес из BX также может быть адресом как байта, так и слова (в приведенном выше примере такой неоднозначности не было, т. к. число 300 может быть только словом). Поэтому с помощью оператора PTR надо указать, операнды какого размера мы имеем в виду:

```
MOV BYTE PTR [BX], 0        ; пересылка байта
MOV WORD PTR [BX], 0        ; пересылка слова
```

Лекция 14-15

Модификация по нескольким регистрам

Мы рассмотрели ситуации, когда модификация адресов осуществлялась по одному регистру-модификатору. Однако идею модификации легко обобщить на случай нескольких модификаторов. Для этого надо в командах вместе с адресом указывать несколько таких регистров. В ПК разрешено указывать сразу два модификатора, причем один из них обязательно должен быть регистром BX или BP, а другой - регистром SI или DI (модифицировать по парам BX и BP или SI и DI нельзя). Возможный пример:

```
MOV AX, A[BX][SI]
```

В данном случае исполнительный адрес вычисляется по формуле:

$$Аисп = (A + [BX] + [SI]) \bmod \mathbf{2^{16}}$$

Модификация по двум регистрам обычно используется при работе с двумерными массивами. Пусть, к примеру, имеется матрица A размером 10x20:

```
A DB 10 DUP(20 DUP(?))  ; A[0..9,0..19]
```

и требуется записать в регистр AL количество таких строк этой матрицы, в которых начальный элемент строки встречается в ней еще раз.

При расположении элементов матрицы в памяти по строкам (первые 20 байтов - начальная строка матрицы, следующие 20 байтов - вторая строка и т. д.) адрес элемента A[i,j] равен A+20*i+j. Для хранения величины 20*i отведем регистр BX, а для хранения j - регистр SI. Тогда A[BX] - это начальный адрес i-й строки Матрицы, а A[BX][SI] - адрес j-го элемент этой строки:

```
        MOV AL, 0               ; количество искомых строк
; внешний цикл (по строкам)
        MOV CX, 10          ; счетчик внешнего цикла
        MOV BX, 0               ; смещение от A до начала строки (20*i)
L:      MOV AH, A[BX]           ; AH – начальный элемент строки
        MOV DX, CX              ; спасти CX внешнего цикла
; внутренний цикл (по столбцам)
        MOV CX, 19          ; счетчик внутреннего цикла
        MOV SI, 0               ; индекс элемента внутри строки (j)
```

```
L1:     INC SI                  ; j:=j+1
        CMP A[BX][SI], AH       ; A[i,j]=AH?
        LOOPNE L1                     ; цикл, пока A[i,j]<>AH, но не более 19 ;
                                      раз
        JNE L2                        ; AH – не повторился --> L2
        INC AL                        ; учет строки
; конец внутреннего цикла
L2:     MOV CX, DX                    ; восстановить CX для внешнего цикла
        ADD BX, 20              ; на начало следующей строки
        LOOP L                        ; цикл 10 раз
```

Запись модифицируемых адресов

Пусть A обозначает адресное выражение, а E - любое выражение (адресное или константное), тогда в языке ассемблера допустимы следующие три основные формы записи адресов в командах, которые задают следующие исполнительные адреса:

A : Аисп=A

E[M] : Аисп=(E+[M]) mod $\mathbf{2^{16}}$ (M: BX, BP, SI, DI)

E[M1][M2] : Аисп=(E+[M1] + [M2]) mod $\mathbf{2^{16}}$ (M1: BX, BP; M2: SI, DI)

(Замечание: если E=0, то 0 можно опустить: 0[M] = [M].)

В ПК в качестве регистра-модификатора можно использовать не любой регистр, а только один из следующих четырех: BX, BP, SI или DI. При модификации только по одному регистру модификатором может быть любой из этих четырех регистров. Однако при модификации по двум регистрам в ПК разрешается указывать не любую пару регистров-модификаторов, а только такую, где один регистр - это BX или BP, а другой регистр - SI или DI.

Следующие записи эквивалентны:

[x][y] = [x]+[y] = [x+y]

Например:

A[BX][DI], A[BX+DI], [A+BX+DI], A[BX]+[DI]

Команда LEA

Загрузка исполнительного адреса (load effective address): LEA rl6, A

Эта команда вычисляет исполнительный адрес второго операнда и записывает его в регистр r16: r16:=Аисп. Флаги команда не меняет. В качестве первого операнда может быть указан любой регистр общего назначения, а в качестве второго - любое адресное выражение (с модификаторами или без них).

Одним из примеров, где полезна команда LEA, является вывод строки по операции OUTSTR. Эта операция, требует, чтобы начальный адрес выводимой строки находился в регистре DX. Так вот, засылку этого адреса в DX и следует делать по команде LEA.

```
S DB ‘a+b=c’, ‘$’
...
LEA DX, S               ; DX:=адрес S
OUTSTR                  ; будет выведено: a+b=c
```

Особенности использования:

```
Q DW 45
LEA BX, Q               ; BX:=адрес Q
MOV BX, Q               ; BX:=содержимое Q (=45)
```

или

```
MOV BX, 50
LEA CX, [BX+2]          ; CX:=[BX]+2=50+2=52
LEA DI, [DI-3]          ; DI:=DI-3 (но понятнее: SUM DI, 3)
```

Команда XLAT

Это так называемая команда перевода, перекодировки:

Перекодировка (translate): XLAT

Действие этой команды заключается в том, что содержимое байта памяти, адрес которого равен сумме текущих значений регистров BX и AL, записывается в регистр AL: AL:=байт по адресу [BX+AL]. Флаги не меняются. Команда XLAT используется для перекодировки символов.

```
DIG16 DB ‘0123456789ABCDEF’  ; DIG16[0..15]
LEA BX, DIG16                ; BX – на начало DIG16
XLAT                         ; AL:=DIG16[AL]
```

Структуры

Структура - это составной объект, занимающий несколько соседних ячеек памяти. Компоненты структуры называются полями, они могут быть разного типа (размера): например, одно поле может быть байтом, другое - словом и т. д. Поля именуются, доступ к полям осуществляется по именам. Описание структуры выглядит так:

```
<имя типа> STRUC
        <описание поля>
        ...
        <описание поля>
<имя типа> ENDS
```

Пример:

```
DATE STRUC
  Y DW 1994
  M DB 3
  D DB ?
DATE ENDS
```

После того как описан тип структуры, можно определять в программе переменные этого типа, отводить под них память (под описание типа память не отводится). Такие переменные называются переменными-структурами или просто структурами. Описываются они с помощью директив следующего вида:

```
имя_переменной имя_типа <нач_знач {, нач_знач}>
```

Пример:

```
DT1 DATE <?,6,9>
DT2 DATE <1998,,>
DT3 DATE <,,>
```

Правила задания начальных значений для полей следующие:

- если в качестве начального значения указан знак ?, то соответствующее поле не получит никакого начального значения (таково, например, поле Y в переменной DT1);
- если в качестве начального значения указано выражение или строка, то значение этого выражения или сама строка становится начальным значением поля (таковы, например, поля M и D в переменной DT1 и поле Y в переменной DT2);
- если же начальное значение не указано (указано "пусто"), то возможны два случая:

если при описании типа структуры в этом поле указано какое-то значение по умолчанию, то оно и становится начальным значением этого поля данной переменной (таково, например, поле M в переменных DT2 и DT3);

если же в описании типа для этого поля не указано значения по умолчанию, то это поле данной переменной не получит никакого начального значения (таково, например, поле D в переменной DT3).

Отметим, что при задании начальных значений полей возможно следующее сокращение: если в уголках не указываются начальные значения для нескольких последних полей, т. е. в конце стоит несколько запятых подряд, то эти запятые можно опустить. Например:

DT2 DATE <1998,,>   эквивалентно DT2 DATE <1998>
DT3 DATE <,,>   эквивалентно DT3 DATE <>

Одной директивой можно описать сразу несколько структур, т. е. можно описать массив, элементами которого являются структуры. Для этого в директиве указывается несколько операндов и/или конструкция повторения DUP. Например

DTS DATE <,12,5>, 10 DUP (<>)

Описав тип структуры и переменные этого типа, мы получаем право работать с этими переменными-структурами. Как единое целое структуры обрабатываются довольно редко, обычно они обрабатываются по полям. Так вот, чтобы сослаться на поле структуры, надо использовать конструкцию вида:

<имя переменной-структуры>.<имя поля>

Например: DT3.D

Оператор ‘.’

Точка, указываемая в ссылке на поле структуры, - это на самом деле один из операторов ЯА, обращение к которое в общем случае имеет такой вид:

<адресное выражение>.<имя поля структуры>

Этот оператор относится к адресным выражениям и обозначает адрес, вычисляемый по формуле:

<адресное выражение>+<смещение поля в структуре>

Например:

DT1.D = DT1+D = DT1+3

TYPE (DT1.D) = TYPE D = BYTE

Нескалярные поля структур

Как известно, в директивах DB, DW и DD можно указывать несколько операндов и конструкцию повторения DUP. Подобные директивы можно использовать и при описании полей структур. Например, допустимо такое описание типа структуры:

```
 STUD STRUC                    ; студент
  FAM DB 10 DUP(?)             ; фамилия
 NAME DB ‘****’           ; имя
   GR DW ?                ; группа
MARKS DB 5, 5, 5          ; оценки
      ENDS
```

Здесь на поле FAM отводится 10 байтов, на NAME - 4 байта, на GR - 2 байта и на MARKS - 3 байта.

Так вот, если в директиве, описывающей поле, имеется более одного операнда или конструкция повторения, то при описании переменной данного типа для этого поля нельзя указывать начальное значение, нельзя указывать и знак ? - соответствующая позиция в уголках должна быть пустой. Из этого правила есть только одно исключение: если поле описано как строка, то такому полю можно давать начальное значение, но оно обязано быть также строкой, причем равной или меньшей длины (при меньшей длине добавляются пробелы справа).

Примеры:

```
S1 STUD <’Иванов',  ...>     ; нельзя (поле FAM - не строка)
S2 STUD <,'Оля',101,>        ; можно (S2.NAME='0ля’, S2.GR=101)
S3 STUD <,'Ольга'>                 ; нельзя (5 символов вместо 4)
```

Причина подобных ограничении в том, что если разрешить задавать начальные значения для нескалярных полей, то будет путаница в том, какое значение к какому полю относится.

ПРИЛОЖЕНИЕ 2

ЛАБОРАТОРНАЯ РАБОТА №1

АССЕМБЛИРОВАНИЕ И ОТЛАДКА ПРОГРАММ. ИЗУЧЕНИЕ СИСТЕМЫ КОМАНД И СПОСОБОВ АДРЕСАЦИИ ОПЕРАНДОВ.

1. ЦЕЛЬ РАБОТЫ:

- Знакомство с полным циклом создания ассемблерной программы.
- Изучение системы команд и способов адресации операндов процессоров **i80x86** в реальном режиме с помощью отладчика **Turbo Debugger**.

2. ЭТАПЫ СОЗДАНИЯ ПРОГРАММЫ.

TURBO ASSEMBLER фирмы *BORLAND (TASM)* поддерживает два синтаксических стандарта или режима:

- *MASM* (по умолчанию), совместимый с макроассемблером фирмы *Microsoft*
- *IDEAL*, режим улучшенного синтаксиса фирмы *Borland.*

Разработка программы на языке ассемблера включает четыре этапа:

- **1-ый этап**. Подготовка исходного текста программы и оформление его в виде текстового файла (одного или нескольких) с помощью какого-нибудь редактора в формате **DOS** с расширением **ASM.**
- **2-ой этап**. Ассемблирование программы с применением транслятора **TASM**. В результате трансляции получаем объектный файл с расширением **OBJ**. Если программа состоит из нескольких файлов (модулей), то их ассемблирование производится независимо друг от друга. Если в процессе трансляции будут обнаружены ошибки, то объектный файл не создаётся, а формируется сообщение об ошибках. Ошибки устраняются, после чего трансляция повторяется. Объектный файл (двоично-кодированное представление программы) не может быть запущен на исполнение, так как в нём не содержится информация о загрузке сегментов программы в памяти компьютера.
- **3-й этап.** Компоновка программы производится компоновщиком (редактором связей) **TURBO LINKER** и заключается в объединении объектных модулей в один исполняемый файл с назначением стартового адреса программы. Исполняемый файл имеет расширение **EXE**.
- **4-ый этап** состоит в отладке программы с использованием отладчика **TURBO DEBUGGER**, который будет являться основным инструментом при изучении системы команд и адресации операндов в памяти в процессе выполнения данной работы.

Войдите в текстовый редактор (один из тех редакторов, которые формируют файлы в коде ASCII) и введите следующие строки программы под названием HELLO.ASM:

```
                IDEAL
                MODEL small
                STACK 256
                DATASEG
ExCode          DB 0
```

```
Promt            DB 'Это время после полудня?(Да/Нет)-[Y/N]$'
GoodMorning      DB 13,10,'Доброе утро!',13,10,'$'
GoodAfternoon    DB 13,10,'Здравствуйте!',13,10,'$'
                 CODESEG
Start:           mov ax,@data          ;Установка в ds адреса сегмента
                 mov ds,ax                    ;данных
                 mov dx,OFFSET Promt          ;Сообщение-запрос
                 mov ah,9                     ;Функция Dos вывода сообщения
                 int 21h               ;на экран
                 mov ah,1                     ;Функция Dos ввода символа с
                 int 21h               ;клавиатуры
                 cmp al,'Y'                   ;Y-прописная
                 jz IsAfternoon        ;да, время после полудня
                 cmp al,'y'                   ;y- строчная
                 jz IsMorning                 ;нет, до полудня
IsAfternoon:     mov dx,OFFSET GoodAfternoon    ; Указание на "Здравствуйте"
                 jmp SHORT Disp               ;Указание на "Доброе ;утро"
IsMorning:       mov dx,OFFSET GoodMorning
Disp:            mov ah,9                     ;Функция Dos вывода сообщения на
                 int 21h               ;экран
Exit:            mov ah,4ch                   ;Функция DOS- выход из программы
                 mov al,[ExCode]              ;Возврат кода ошибки
                 int 21h               ;Вызов DOS. Останов прогр.
                 END Start                    ;Конец программы/точка входа
```

Для ассемблирования файла Hello.asm в командной строке наберите:

tasm hello.asm

и нажмите клавишу *Enter*. Будет создан объектный файл с этим же именем *hello.obj*. Заметим, что расширение имени файла *asm* вводить не требуется, т.к. **tasm** принимает это по умолчанию. На экране вы увидите следующее:

Turbo Assembler Version X.X Copyright (C) 1988, 19XX by Borland International Inc.

Assembling file: Hello.asm

Error messages: None

Warning messages: None

Passes 1

Remaining memory: ***K

Из сообщения следует, что ассемблирование завершено без ошибок и отсутствия предупреждений. Предупреждение не является ошибкой, однако, его игнорирование может привести к неприятностям в дальнейших этапах работы с программой, поэтому лучше своевременно реагировать на данный тип замечаний.

Для компоновки программы введите командную строку *tlink hello.obj*

Здесь, также как и при ассемблировании, расширение имени *obj* не является обязательным. По завершению компоновки будет сформирован файл *hello.exe* с выводом на экран сообщения.

Turbo Linker Version 5.1 Copyright (C) 1988, 1991 by Borland International Inc.

Теперь программу *hello.exe* можно запустить на исполнение, результатом которого будет вывод на экран сообщения:

Это время после полудня? (Да/Нет)-[Y/N]

Курсор будет мерцать после последнего символа в ожидании ввода ответа. Введите прописную букву *Y*. Программа ответит:

Здравствуйте!

Если будет введена строчная буква y, то программа ответит:

Доброе утро!

3.ОПЦИИ КОМАНДНОЙ СТРОКИ.

В процессе ассемблирования или компоновки вы можете выбирать различные исполнения процесса, которые задаются опциями в командной строке **TASM** или **TLINK**. Для вывода списка опций командной строки **Turbo Assembler** наберите *tasm* и нажмите <*Enter*>.Для вывода списка опций командной строки **Turbo Linker** наберите *tlink* и нажмите <*Enter*>.

Опции описываются одной или несколькими буквами. Для задания опции наберите дефис (или косую черту) и соответствующую букву между командой *tasm* или *tlink* и именем программы, которую вы ассемблируете либо компонуете. Например, для ассемблирования программы *Hello.asm* и получения файла с листингом (файл, в котором содержится описание процесса ассемблирования), используйте команду: *tasm/l hello*

Опции разрешается набирать как прописными, так и строчными буквами. Исполните эту команду и затем рассмотрите файл с листингом *Hello.lst*, используя текстовый редактор. В листинге, каждая строка начинается с номера, затем следует байты объектного кода и, наконец, собственно текст программы. Кроме того, **TASM** выводит в файле листинга таблицы, где содержится информация о метках и сегментах, включая значение и тип каждой метки, и атрибуты каждого сегмента. При ассемблировании программы вы можете использовать в одной командной строке несколько опций, разделяя их пробелами (что не является обязательным).

tasm/l/c hello

tasm-l-c hello

tasm/zi hello

Первая команда дополняет файл листинга таблицей перекрёстных ссылок, в которой указывается, где была определена каждая метка и где на неё есть ссылка. Вторая команда делает то же самое, и просто показывает другую форму записи. Третья команда добавляет в *Hello.obj* информацию, необходимую при последующем использовании **Turbo Debugger**.

Turbo Linker также имеет различные опции командной строки, определяемые аналогичным образом:

tlink/v hello

tlink/m/l hello

tlink/x hello

Опция *v* в командной строке подготавливает *hello.exe* для использования его с **Turbo Debugger**. Следующая команда содержит две опции, которые приводят к созданию расширенного файла отображения или файла загрузки *Hello.map* и добавляет в этот файл вспомогательную информацию о номерах строк (*l*). Файл загрузки содержит список сегментов программы, стартовый адрес программы, предупреждения и сообщения об ошибках во время работы компоновщика. Опция */x* запрещает **TLINK** создавать файл отображения.

4. РЕАЛЬНЫЙ РЕЖИМ ПРОЦЕССОРОВ I80X86. СЕГМЕНТИРОВАННАЯ МОДЕЛЬ ПАМЯТИ

Физическая память, к которой процессор имеет доступ по шине адреса, организована как последовательность ячеек - байтов. Каждому байту соответствует свой уникальный адрес (его номер), называемый *физическим*. Диапазон значений физических адресов зависит от разрядности шины адреса, и так, например, для процессора **Pentium** □□ составляет от 0 до 2^{36} - 1 (64 Гбайт). Все программы, написанные для процессора i8086, могут исполняться на процессорах Intel старших поколений в так называемом *реальном режиме* (R - режим). В реальном режиме происходит формирование 20-разрядного физического адреса, аналогичное процессору i8086. По умолчанию, в реальном режиме используется *сегментная модель* памяти с объёмом сегмента в 64 Кбайта, расположение объекта в котором, относительно его базового адреса (см. рис. 1.1.), определяется смещением OFFSET.

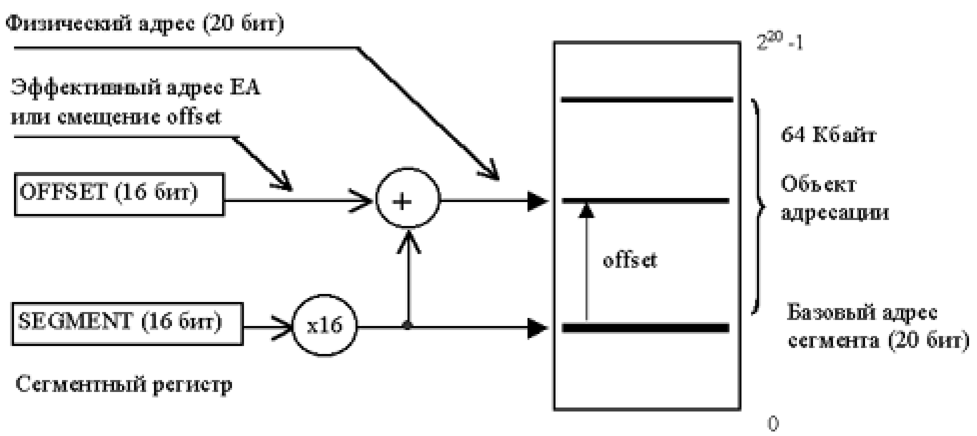


Рис. 1.1. Сегментированная модель памяти реального режима.

Независимо оттого, что процессор формирует всегда физический адрес, программист имеет дело с логическим адресом, включающим две компоненты:

"SEGMENT (сегмент)**:** OFFSET (смещение)".

Как сегмент, так и смещение представляются 16-разрядными словами. Сегментация - механизм адресации, обеспечивающий несколько независимых адресных пространств как в пределах одной задачи, так и в системе в целом для защиты задач от взаимодействия друг на друга. Назначением базовых адресов сегментов занимается операционная система (ОС), а внутри каждого сегмента адреса формируются программой. Это эффективный адрес или смещение OFFSET. Важные замечания:

- **ОС в зависимости от объёма кода и данных программы может так назначить базовые адреса, что они будут перекрываться;**
- **Физический адрес базы сегмента кратен числу 16, поэтому сегменты начинаются на границах блоков с разницей в 16 байт (параграф);**
- **Сегментная организация обеспечивает создание позиционно - независимых или динамически перемещаемых программ. Каждое исполнение программы может происходить с различными значениями базовых адресов сегментов, которые выбираются ОС исходя из особенностей загрузки памяти.**

5. СТРУКТУРА ПРОГРАММЫ ДЛЯ *EXE*-файлов И ЕЁ ОБРАЗ В ПАМЯТИ.

Программы, выполняемые под управлением DOS, могут принадлежать к одному из двух типов, которым соответствуют расширения имён программных файлов *.COM* и *.EXE*. Основное различие этих программ заключается и том, что программы типа *.COM* состоят из единственного сегмента объёмом в 64 Кбайт, в котором размещаются программные коды, данные и стек, а в программах типа *.EXE* для каждого из них выделяются отдельные сегменты. Управление сегментами - один из наиболее сложных аспектов программирования на языке ассемблера. При этом ассемблер имеет не один, а целых ***два набора директив управления сегментами***. Первый набор, включающий упрощенные директивы определения сегментов, позволяет достаточно просто ими управлять и идеально подходит для компоновки ассемблерных модулей с языками высокого уровня (Pascal и С). Второй набор, включающий стандартные директивы определения сегментов, предназначен для построения сложных ассемблерных программ.

Мы будем использовать упрощенные директивы, введение которых возможно лишь с одновременным указанием используемой ***модели памяти***. В свою очередь, модель памяти неявно задаёт атрибуты упрощенных директив, определяющих действия компоновщика Turbo Linker при формировании исполнительного файла программы. Ассемблер использует те же модели памяти, что и языки высокого уровня. Наибольшее применение в ассемблерных программах с расширением *.EXE* нашла малая модель памяти ***Model small***, предусматривающая размещение структурных частей программы в двух сегментах:

- Сегмент кода программы (64 Кбайт),
- Сегмент данных и стека (64 Кбайт).

Рассмотрим шаблон программы для *EXE*-файлов.

%Title "Оболочка для *EXE*-файлов"

```
Ideal         ; Переводит Turbo Assembler в режим Ideal
P486N         ; Разрешает инструкции старших поколений процессоров.
              ; При её отсутствии действует по умолчанию P8086.
              ; Разрешает использование 32 разрядных регистров для
              ; адресации и хранения данных
Model small   ; Директива описания модели памяти.
              ; Вводит упрощенные директивы управления сегментами.
Stack 256     ; Резервирует пространство для стека программы
              ; (значение в байтах, следующее за директивой).
```

1. Здесь следует располагать макроопределения EQU и =.
2. Вставьте здесь директиву Include "filename".

```
Dataseg       ; Начало сегмента инициализированных данных.
              ; Допускается также расположение здесь и
              ; неинициализированных данных, начальные значения которых
              ; неизвестны на момент запуска программы.
```

1. Здесь описываются переменные с помощью директив DB,DW и т. п.
2. Здесь опишите все переменные типа EXTRN

```
CODESEG       ; Начало сегмента кода, т.е. части программы, содержащей
              ; команды процессора.
```

Здесь определите все подпрограммы типа EXTRN

```
Start:                          ; Точка начала исполнения программы.
        mov ax, @data           ; Установка в регистре DS адреса сегмента данных
        mov ds,ax
```

Здесь располагается программа, вызовы подпрограмм и т. п.

```
Exit:   mov ax,4C00h            ; Функция DOS: Выход из программы.
        Int 21h                 ; Вызов DOS. Останов программы.
        END Start               ; Директива конца программы/точки входа.
                                ; Текст, расположенный в файле после этой строки
                                ; будет игнорироваться ассемблером.
```

При использовании директивы *Model* ассемблер делает доступным несколько предопределённых (служебных) идентификаторов, к которым можно обращаться во время работы программы, чтобы получить информацию об адресах используемых сегментов. Например:
@code - 16-разрядный адрес сегмента кода,
@data - 16-разрядный адрес сегмента данных с именем *dgroup,*
@stack - 16-разрядный адрес сегмента стека

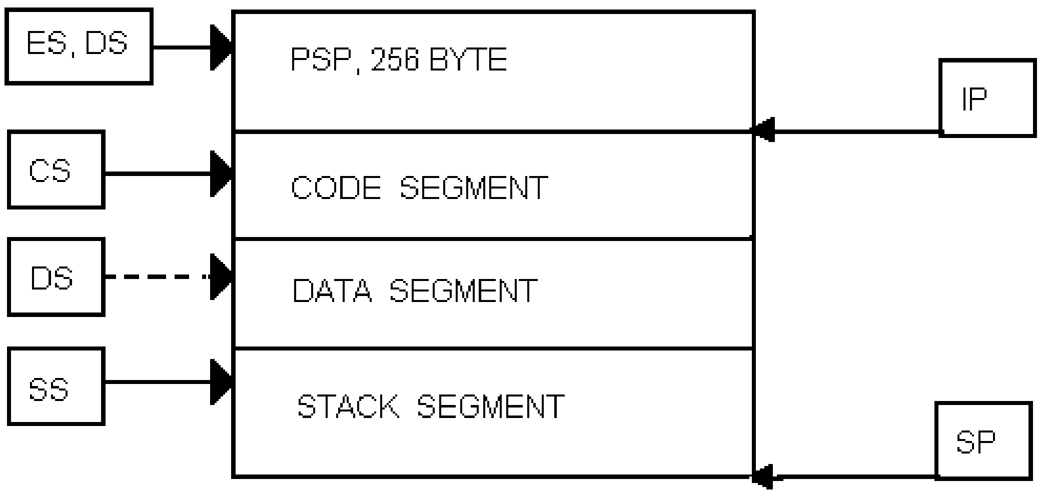


Рис. 1.2 Образ памяти программы .EXE

Для того чтобы понять, почему кодовый сегмент начинается с загрузки сегментного регистра данных *DS*, а не с загрузки сегментного регистра *CS*, необходимо рассмотреть функции DOS при загрузке программы в память.

Образ программы в памяти, представленный на рис.1.2, начинается с префикса программного сегмента *PSP* (Program Segment Prefix). *PSP* всегда имеет размер 256 байтов и содержит данные, используемые операционной системой в процессе исполнения программы.

Вслед за *PSP* располагаются сегменты программы. Сегментные регистры *es* и *ds* автоматически инициализируются на начало *PSP*. Это даёт возможность, при сохранении значения одного из регистров, обращаться к нему в случае необходимости. В указатель команд *ip* загружается относительный адрес точки входа в программу (операнд директивы *END*), а в указатель стека *sp* -смещение конца сегмента стека.

Таким образом, после загрузки программы в память адресуемыми являются все сегменты, кроме сегмента данных. Это и осуществляется первой парой команд программы с использованием предопределённого идентификатора @data.

Завершение программы означает передачу управления командному процессору DOS - Command.com, который выводит на экран системный запрос в ожидании следующей команды от оператора. Делается это с помощью функции DOS (*AH=4Ch*). Эта функция требует единственного параметра - кода возврата, который и помещается программой в регистр *al*. Если этот код равен нулю, то исполнение программы прошло корректно, в противном случае была допущена ошибка в предшествующих обращениях программы к другим системным функциям DOS.

6. РАБОТА С ОТЛАДЧИКОМ TURBO DEBUGGER (TD)

TASM умеет ассемблировать синтаксически правильные программы, но не понимает, что, собственно эта программа делает. Часто программа работает не так, как, по вашему мнению, должна была бы работать. В такой ситуации может помочь *TD*-программа, разработанная для поиска и исправления логических ошибок. Подобно всем отладчикам, *TD* может работать в режиме супервизора, беря на себя управление программой в режиме пошагового исполнение кода программы. Можно при этом изменять значения операндов в памяти, а также значения регистров и флагов. *TD* используется и в качестве учителя для изучения форматов машинных команд процессора в различных режимах адресации операндов.

Чтобы показать, как использовать *TD* при изучении языка ассемблера, исследуем программу *Hello* под управлением отладчика. Произведём заново ассемблирование и компоновку программы с опциями, которые добавляют отладочную информацию в *obj*- и *exe*-файлы:

tasm/zi hello.asm

tlink/v hello.obj

td hello.exe

После выполнения последней команды на экране увидите окно *Module TD* с исходным текстом программы *Hello.asm*. Это окно номер 1. Просмотреть программу можно, используя клавиши управления курсором, передвигая его вверх и вниз по тексту.

Для получения другого представления программы войдите в меню *Viev* [обзор], выберите команду *CPU* и нажмите <*F5*> для распахивания окна на весь экран. В *CPU*-окне, состоящем из пяти областей, содержится в сокращённом виде исходный текст вашей программы, действительные машинные коды, находящиеся в памяти, значения регистров и флагов, стек и дамп байтов памяти.

Для передвижения курсора из одной области в другую нажимайте клавишу <*Tab*>. Область окна *CPU,* в которой находится курсор, считается активной. Нажатие клавиши <*Alt+F10*> вызывает появление локального меню для активной секции окна *CPU*.

Перейдите в главную область окна *CPU* и нажмите <*Alt+F10*>. Выберете команду *Mixed* (смесь), которая имеет три установки: *No, Yes* и *Both*. Режим *Both (*оба) устанавливается по умолчанию и является наилучшим способом просмотра, показывая в левой колонке байты машинного кода, а в правой - строки исходного текста программы.

Окно *CPU* используется для наблюдения за текущим состоянием процессора при пошаговом выполнении инструкций программы. Маленькая стрелка-треугольник слева от первой команды *mov ax,@data* показывает, что она является следующей исполняемой командой. Для выполнения этой команды нажмите <*F8*>, стрелка перейдёт на следующую команду с изменённым значением регистра *ax* в окне регистров. Снова нажмите <*F8*>, для выполнения инструкции *mov ds,ax*. Обратите внимание, что значение в регистре *ds* стало таким же, как и в регистре *ax,* но произошло изменение сегментации дампа памяти (левая нижняя область) с регистра *ds* на регистр *es*.

Сделайте активной область с дампом памяти, нажмите <*Alt+F10*> и в появившемся локальном меню выберите команду *Goto* (переход). Появится заставка, в которой наберите *ds*:0000 и нажмите <*Enter*>. Теперь дамп памяти будет соответствовать массиву инициализированных данных вашей программы при их побайтовом представлении (*ASCII*-код символов переменной *PROMT, Good Morning и Good Afternoon*). Для просмотра всего дампа используйте передвижение курсора с помощью стандартных клавиш. Объяснение этому факту следует из особенностей загрузки операционной системой MS-DOS в память программ с расширением *EXE* (рис.1.2).

Чтобы лучше разобраться с представлением переменной *PROMT* в дампе памяти, войдите в меню *Data* (данные), выберите команду *Inspect* (проверка). В появившейся заставке наберите *PROMT* и нажмите <*Enter*>. Раздвиньте с помощью мыши появившееся окно *Inspecting Promt* по вертикали на всю высоту экрана в правой его части. В левой колонке данного окна указывается номер элемента переменной массива *PROMT[i]*, а в правой значение *ASCII*-кода этого элемента. Найдите эти коды в дампе памяти, а именно:

ds:0000 exCode=00h
ds:0001 Promt[0]='Э'=9Dh
ds:0002 Promt[1]='т'=E2h
ds:0003 Promt[2]='о'=AEh

и т. д.

Продолжим покомандное исполнение программы с помощью клавиши <*F8*>, предварительно закрыв окно *Inspecting Promt*, щёлкнув мышью по кнопке закрытия. После выполнения каждой команды наблюдаем за изменением значений регистров.

Описанный выше процесс выполнения программы прерывается после исполнения команды прерывания DOS (*Int* 21,функция *ah*=1) по вводу символа с клавиатуры на экран. ***TD*** переводит вас с окна ***CPU*** в окно пользователя *User Screen*, в котором появляется сообщение-запрос переменной ***PROMT***. Ответив на этот запрос (прописной или строчной буквой: *Y* или *y*), вы автоматически снова перейдёте в окно ***CPU***, для дальнейшего покомандного выполнения программы.

Переход, при необходимости, к окну пользователя от ***TD*** и обратно можно осуществить нажатием клавиш <*Alt+F5*>. Если вы забыли это сочетание клавиш, то обратитесь к меню управления окнами ***WINDOW***, найдите строку ***User Screen***, с уже упомянутыми клавишами. Кстати, ниже этой строки будет приведён список всех открытых вами окон *TD*, а именно:

Module Hello (1)
CPU (2)
Inspector (3)

Вы можете вызвать любое открытое вами окно с помощью нажатия клавиш <*Alt+ номер окна*>, или путём их последовательного перебора с помощью клавиши <*F6-Next*>.

Если вы хотите закрепить данное расположение окон при следующей работе с *TD*, то войдите в меню *OPTIONS* (параметры), выберите кнопку с командой *Layout* (схема окон). Данная схема будет зафиксирована утилитой ***Tdconfig.dt***. Раз уж мы находимся в меню ***OPTIONS***, укажем здесь ещё на одну полезную команду *Display options* (вывести параметры). Всплывающая заставка *Display swapping* (переключение экрана) позволяет выбрать один из трёх способов управления переключением между экраном *TD* и экраном пользователя. По умолчанию устанавливается параметр *Smart* (эффективный), который позволяет переключиться на экран пользователя ***User Screen*** автоматически по требованию программы.

Вернёмся снова в окно ***CPU*** для завершения работы с программой *Hello*. Используя клавишу <*F8*>, доводим маркер исполнения текущей команды до команды с меткой *Exit* (вызов функции DOS с номером 4Ch для выхода из программы) и, нажав <*Alt+F5*>, увидим на экране пользователя ответ программы на наш диалог с ней.

Если вы хотите повторить выполнение программы, то она может быть перезагружена с произвольной команды с помощью нажатия клавиш <*Ctrl+F2*> или командой меню *RUN = Program Reset* (сброс программы). При этом программа снова загружается с диска и *TD* восстанавливает свои исходные опции. Если вы находитесь в окне *CPU* или ***Module***, то на дисплее не будет показан возврат к началу вашей программы - для этого надо нажать <*F8*>.

Завершая краткое описание работы с отладчиком *TD*, перечислим здесь возможные режимы исполнения программы в *TD*:

- Режим автоматического исполнения, - клавиша *F9 (Run).*
- Выполнение по шагам (клавиши *F8* или *F7-Trace into*). Отличие в назначении этих клавиш проявляется в том, что клавиша *F7* используется для пошагового исполнения тела цикла, процедуры или подпрограммы обработки прерывания. Клавиша же *F8* исполняет эти процедуры как одну обычную команду и передаёт управление следующей команде программы. Используйте клавишу *F7* для просмотра программы *LoopCall*, которая имеет указанные выше процедуры.
- Выполнение до текущего положения курсора. Для активизации этого режима необходимо установить курсор на нужную строку программы (строка будет подсвечиваться другим цветом) и нажать клавишу *F4*.
- Выполнение с установленными точками прерывания (*Breakpoints*). Перед исполнением программы необходимо установить эти точки, для чего следует перейти в нужную строку программы и нажать клавишу ***F2*** **(*toggle*).** Выбранная строка с контрольной точкой подсвечивается красным цветом. Чтобы убрать контрольную точку, надо повторить эту операцию снова. После установки точек прерывания программа запускается на исполнение кл. *F9*. После первого нажатия кл. *F9* программа остановится на первой точке прерывания, после второго - на второй точке и.т.д. Это очень удобный режим отладки программы, когда необходимо контролировать правильность её исполнения в некоторых характерных точках.

Завершать работу с отладчиком следует командой ***FILE = Quit*** или с помощью клавиш <*Alt+X*>.

TD имеет множество других возможностей, изучить которые вам предлагается самостоятельно при исследовании разрабатываемых вами программ.

5. РЕЖИМЫ АДРЕСАЦИИ И ФОРМАТЫ МАШИННЫХ КОМАНД (см. лекцию 4)

6. КОМАНДА MOV И РЕЖИМЫ АДРЕСАЦИИ КОМАНД

По нижеприведенному листингу программы ознакомьтесь с различными режимами адресации команды ***MOV***

Команда MOV и режимы адресации

```
                IDEAL
                MODEL small
                STACK 256
                Value = 528
                DATASEG
exCode          DB 0
b_x             DB 1,2,4
```

```
w_x         DW 8,16,32,64
Label       b_var   byte
w_var       DW 1234h
            CODESEG
Start:      mov ax,@data; Установка в ds адреса
            mov ds,ax             ; сегмента данных.

; Непосредственная адресация
            mov al,255            ;255=0FFh-беззнаковое число
            mov ah,-1             ;-1=0FFh-отрицательное число
            mov ax,value/5+20     ;Загрузка в ах константного выражения
            mov bx,OFFSET w_x             ;Адрес переменной w_x в bx, bx=0004h

;Регистровая и прямая адресации.
            mov dl,al             ;
            mov al,[b_x]          ;al=b_x[0]=01h.
            mov dx,[w_x]          ;dx=w_x[0]=0008h.
            mov si,[w_var]        ;si=1234h
            mov al,[b_var]        ;al=[Low w_var]=[b_var]=34h
            mov ah,[b_var+1]      ;ah=[High w_var]=[b_var+1]=12h

;Косвенная регистровая.
            mov cx,[bx]           ;cx=w_x[0]=0008h.
            mov [word bx],-2      ;w_x[0]=-2=0FFFEh.

;Базовая адресация.
            mov ax,[bx+2]         ;ax=w_x[1]=16=0010h
            mov [word bx+2],24    ;w_x[1]=24=0018h.

;Индексная адресация.
            mov si,1
            mov al,[si+b_x]       ;al=b_x[1]=2=02h.

;Базово индексная адресация.
            inc si
            mov bx,2
            mov ax,[bx+si+w_x]    ;ax=w_x[2]=32=0020h
            mov [word bx+si+w_x],128  ;w_x[2]=128=0080h

;Применение команды lea.
            lea bx,[w_x+si]       ;bx=OFFSET w_x+si=OFFSET w_x[1]=0006h

;Команды push и pop.
            push bx               ;Сохранить bx и si
            push si         ;в стеке.
            mov bx,10h            ;Установить текстовые
            mov si,20h            ;значения.
            pop si                ;Восстановить из стека
            pop bx          ;сохранённые значения
Exit:       mov ah,04Ch  ;Ф-ция DOS- выход из программы.
            mov al,[exCode]       ;Возврат кода ошибки.
            int 21h         ;Вызов DOS. Останов
```

```
END        Start              ;Конец программы/точка входа
```

ЛАБОРАТОРНАЯ РАБОТА № 2

ТЕМА: *ЦИКЛЫ.*

1. Цель работы:

- Получение навыков организации циклических структур в ассемблерных программах.

2. Циклы в ассемблерных программах

Организовать циклическую структуру в ассемблерной программе можно двумя способами

- используя команды сравнения (CMP) и передачи управления (Jcc)
- используя специальные команды организации циклов (LOOP).

Команда сравнения CMP сравнивает два числа, вычитая второе из первого, также как и команда SUB, но не сохраняя результат. После выполнения команды флаги состояния устанавливаются в соответствии с результатом операции.
Команды перехода можно разделить на две группы:

- команды условного перехода
- команды безусловного перехода.

Безусловный переход – это такой переход, который передает управление без сохранения информации возврата всякий раз, когда выполняется. Ему соответствует команда JMP, имеющая двухбайтное смещение.
Условный переход проверяет текущее состояние регистра флагов, чтобы определить, передать управление или нет. Все условные переходы имеют однобайтовое смещение.
Ниже приведен фрагмент программы, вычисляющей в цикле сумму цифр от 0 до 9.

```
        MOV   AH,0      ;занести в AH первую цифру «0»
        MOV   AL,0      ;подготовить регистр результата
met:    ADD   AL,AH ;прибавить к результату очередную цифру из AL
        INC   AH        ;увеличить AH на единицу
        CMP   AH,10 ;сравнить значение в AH со значением «10»
        JNE   met       ;если AH<>10 то осуществить переход на MET
```

В приведенном примере в качестве счетчика используется регистр AH и необходимо на каждой итерации цикла самостоятельно производить инкремент счетчика и его сравнение с заданной величиной 10. Для упрощения подобных ситуаций в ассемблере предусмотрены специальные команды для организации циклов. Все команды цикла используют регистр CX в качестве счетчика цикла. Простейшая из них – команда LOOP. Она в конце каждой итерации уменьшает содержимое CX на 1 и передает управление на метку (указанную в команде), если содержимое CX не равно 0. Если вычитание 1 из CX привело к нулевому результату, выполняется следующая команда.
Команда LOOPNE (цикл пока не равно) выходит из цикла, если установлен флаг нуля или если в регистре CX получился 0. Команда LOOPE (цикл пока равно) выполняет обратную к описанной проверку флага нуля: цикл здесь завершается, если регистр CX достиг 0 или если не установлен флаг 0.
Ниже приведен фрагмент программы, решающий описанную выше задачу, используя команды организации циклов.

```
        MOV   AH,0      ;занести в AH первую цифру «0»
        MOV   AL,0      ;подготовить регистр результата
```

```
        MOV  CX,10  ;количество суммируемых цифр
met:    ADD  AL,AH  ;прибавить к результату очередную цифру из AL
        LOOP met         ;если CX<>0 то осуществить переход на MET
```

4. Вывод на экран

Вывод информации в ассемблерных программах осуществляется обычно при помощи сервисных функций DOS *(прерывание 21h)* или BIOS *(прерывание 10h)*. Процесс вывода состоит в следующем:

– определенные регистры микропроцессора загружаются выводимой информацией или адресом буфера, содержащего выводимую информацию;
– в регистр AH заносится номер используемой для операции вывода функции;
– инициируется прерывание.

Ниже представлен перечень функций прерывания 21h и 10h, использующихся для вывода информации.

4.1 Прерывание 21h.

Функция 02h

Вывод на дисплей.

Вход: AH=02h
DL=выводимый символ

Выход: нет

Описание: Посылает символ из DL на стандартный вывод. Обрабатывает символ Backspace (ASCII 8), перемещая курсор влево на одну позицию и оставляя его в новой позиции.

Функция 05h

Вывод на принтер.

Вход: AH=05h
DL= символ, записываемый на стандартный принтер

Выход: нет

Описание: Посылает символ в DL на стандартное устройство принтера, обычно LPT1. Команда DOS MODE может перенаправить этот вывод в последовательный порт.

Функция 09h

Выдать строку.

Вход: AH=09h
DS:DX=адрес строки, заканчивающейся символом '$'.

Выход: нет

Описание: Строка, исключая завершающий ее символ '$', посылается на стандартный вывод. Символы Backspace обрабатываются как в функции 02h. Обычно, чтобы перейти на новую строку, включают в текст пару CR/LF (ASCII 0dh и ASCII 0ah). Строки, содержащие '$', можно выдать через 40h (BX=0), которая посылает символ в DL на стандартное устройство принтера, обычно LPT1.

4.2 Прерывание 10h.

Функция 02h

Вход: AH=02h
BH = видео страница
DH,DL = строка, колонка (считая от 0)

Выход: нет

Описание: Устанавливает курсор в позицию DH,DL. Установка курсора на строку 25 делает курсор невидимым.

Функция 09h

Писать символ/атрибут в текущей позиции курсора.

Вход: AH=09h
BH = номер видео страницы
AL = записываемый символ
CX = счетчик (сколько экземпляров символа записать)
BL = видео атрибут (текст) или цвет (графика)

Выход: нет

Описание: Выводит на экран в текущую позицию курсора символ с заданным атрибутом.

<u>Функция 0ah</u>

Писать символ в текущей позиции курсора.

Вход: AH=0ah
BH = номер видео страницы
AL = записываемый символ
CX = счетчик (сколько экземпляров символа записать)

Выход: нет

Описание: Выводит на экран в текущую позицию курсора заданный символ.

<u>Функция 13h</u>

Вывод строки.

Вход: AH=13h
ES:BP – выводимая строка
CX = длина строки (подсчитываются только символы)
DH,DL = позиция (строка, колонка) начала вывода
BH = номер страницы
AL = код подфункции:
0=атрибут в BL; курсор без изменения
1=атрибут в BL; курсор – в конец строки
2=формат строки: char,attr,...; курсор без изменения
3=формат строки: char,attr,...; курсор – в конец строки

Выход: нет

Описание: Выдает строку в позиции курсора. Символы 0dH (CarRet), 0aH (LineFeed), 08H (backspace) и 07H (Beep) трактуются как команды управления и не высвечиваются.

Некоторые функции прерывания 10h используют для вывода атрибут символа. Для адаптеров цветной графики в текстовом режиме атрибут определен следующим образом:

7	6	5	4	3	2	1	0
fgB	background			brt	foreground		

foreground – цвет переднего плана (от 0 до 0fH)
brt – интенсивность: 1=передний план яркий
background – фоновый цвет (от 0 до 7)
fgB – мерцание: 1=передний план мерцает

Видеоадаптер поддерживает следующие цвета:

00H черный	08H темно-серый
01H синий	09H ярко-синий
02H зеленый	0aH светло-зеленый
03H голубой	0bH светло-голубой
04H красный	0cH светло-красный
05H розовый	0dH светло-розовый
06H коричневый	0eH желтый
07H серый	0fH белый

Вычислить значение атрибута можно, используя следующее выражение:
(фон * 16) + передний план + (128 для мерцания)

Приведенный ниже фрагмент программы иллюстрирует процесс вывода строки на экран.

```
MOV   AH,09H                ;Выбор функции прерывания
```

```
MOV  DX,OFFSET STR    ;Занесение в DX адреса выводимой строки
INT  21H
. . .
STR  DB   10,13,'Hello$' ;Описание строки
```

5. Ввод с клавиатуры

Процесс ввода информации в ассемблерных программах осуществляется аналогично выводу:

– в регистр AH заносится номер функции ввода;

– инициируется прерывание, после выполнения которого определенные регистры процессора содержат либо введенную информацию, либо адрес буфера с введенной информацией.

Ниже описан ряд функций прерывания 21h, используемых для ввода информации с клавиатуры.

Функция 01h

Ввод с клавиатуры.

Вход: AH=02h

Выход: AL= символ, полученный с клавиатуры

Описание: Считывает (ожидает) символ со стандартного входного устройства. Отображает этот символ на стандартное выходное устройство (эхо). Ввод расширенных клавиш ASCII (F1-F12, PgUp, курсор и т.п.) требует двух обращений к этой функции. Первый вызов возвращает AL=0. Второй вызов возвращает в AL расширенный код ASCII.

Функция 07h

Нефильтруемый консольный ввод без эха.

Вход: AH=07h

Выход: AL= символ, полученный с клавиатуры

Описание: Считывает (ожидает) символ со стандартного входного устройства и возвращает этот символ в AL. Не фильтрует. Не проверяет на Ctrl-Break, backspace и т.п. Необходимо вызывать дважды для ввода расширенного символа ASCII.

Функция 08h

Консольный ввод без эха.

Вход: AH=08h

Выход: AL= символ, полученный с клавиатуры

Описание: Считывает (ожидает) символ со стандартного входного устройства и возвращает этот символ в AL. При обнаружении Ctrl-Break выполняется прерывание INT 23H. Необходимо вызывать дважды для ввода расширенного символа ASCII.

Функция 0ah

Буферизированный ввод строки.

Вход: AH=0ah

DS:DX=адрес входного буфера (смотри ниже)

Выход: буфер содержит ввод, заканчивающийся символом CR (ASCII 0dH)

Описание: При входе буфер по адресу DS:DX должен быть оформлен следующим образом:

max	?	?	?	?	?	?	…

MAX - максимально допустимая длина ввода (от 1 до 254) При выходе буфер заполнен данными следующим образом:

max	len	T	E	X	T	…	0dh

LEN - действительная длина данных без завершающего CR (здесь – 0dH).

Символы считываются со стандартного ввода вплоть до CR (ASCII 0dH) или до достижения длины MAX-1. Если достигнут MAX-1, включается консольный звонок для каждого очередного символа, пока не будет введен возврат каретки CR (нажатие Enter). Второй байт буфера заполняется действительной длиной введенной строки, не считая завершающего CR. Последний символ в буфере – всегда CR (который не засчитан в байте длины). Символы в буфере (включая LEN) в момент вызова используются как «шаблон». В процессе ввода действительны обычные клавиши редактирования: Esc выдает "\" и начинает с начала, F3 выдает буфер до конца шаблона, F5 выдает "@" и сохраняет текущую строку как шаблон, и т.д. Большинство расширенных кодов ASCII игнорируются. При распознавании Ctrl-Break выполняется прерывание INT 23H (буфер остается неизменным).

<u>Функция 0ch</u>

Ввод с очисткой

Вход: AH=0ch

AL= номер функции ввода (01H, 06H, 07H, 08H или 0aH)

Выход: нет

Описание: Очищает буфер опережающего ввода стандартного ввода, а затем вызывает функцию ввода, указанную в AL. Это заставляет систему ожидать ввод очередного символа. Следующие значения допустимы в AL: 01H Ввод с клавиатуры; 06H Ввод с консоли; 07H Нефильтрующий без эха; 08H Ввод без эха; 0aH Буферизованный ввод. Приведенный ниже фрагмент программы иллюстрирует буферизированный ввод строки:

```
MOV  AH,0AH       ;занесение в AH номера функции
LEA  DX,BUF       ;загрузка DX адресом буфера BUF
INT  21H
. . .
BUF   DB    30,00,30 DUP ('$'),’$’
```

При организации буфера BUF (размер буфера 30 байт), он весь заполняется символами «$», что удобно, если введенную строку в дальнейшем необходимо выводить на экран или в файл (при условии, конечно, что для ввода буфер будет использоваться лишь однократно).

Задание:

- Разработать программу, ввода строковых данных с клавиатуры.
- В веденной строке удалить пробелы, все строчные символы заменить на прописные
- Вывести результирующую строку на экран.
-

ЛАБОРАТОРНАЯ РАБОТА № 3

ТЕМА: ВВОД ЧИСЕЛ. ПЕРЕВОД ЧИСЕЛ В РАЗЛИЧНЫЕ СИСТЕМЫ СЧИСЛЕНИЯ.

3. Цель работы:

- научиться вводить в ассемблерную программу числовую информацию;
- разработка алгоритмов для перевода чисел в различные системы счисления.

4. Ввод числовой информации.

Ввод числовой информации в ассемблерную программу обычно осуществляется в два этапа:

- ввод строки содержащей число;
- перевод строки в число.

Ввод строк рассматривался в предыдущей лабораторной работе.

Для разработки алгоритма перевода введенной строки в число проанализируем структуру числа в позиционной системе счисления (в такой системе счисления вес цифры определяется ее местоположением в числе):

$2398=2*1000+3*100+9*10+8=2*10^3+3*10^2+9*10^1+8*10^0$

Таким образом, для перевода строки в число из введенной строки «2398» необходимо последовательно выделять цифры и производить суммирование произведений этих цифр и множителей соответствующих позиции цифры в числе. Если буфер для ввода строки был организован, например, следующим образом:

```
BUF    DB      05,00,05 DUP (?)
```

то после ввода строки «2398» он будет выглядеть (в шестнадцатеричной системе счисления) так:

05,04,32,33,39,38,0d

где первый байт – размер буфера, второй – количество введенных символов (без завершающего символа CR), третий, четвертый, пятый, шестой и седьмой – коды символов «2», «3», «9», «8» и «CR» соответственно. Легко заметить, что для того чтобы из кода цифры получить саму цифру необходимо из соответствующего кода вычесть 30h (шестнадцатеричный код нуля). Затем, последовательно в цикле (второй байт – количество введенных символов) выбирая цифры, формировать соответствующий множитель, вычислять произведение и производить суммирование. Нижеследующий фрагмент программы иллюстрирует описанный алгоритм (символы рассматриваются справа налево).

```
;Ввод числа в виде строки
        MOV  AH,0AH        ;в AH номер функции
        LEA  DX,BUF        ;DS:DX адрес буфера для ввода
        INT  21H
;Перевод строки в число, результат в DI
        MOV  DI,0
        LEA  BX,BUF+1      ;в BX адрес второго элемента буфера
        MOV  CX,[BX]       ;в CX количество введенных символов
        XOR  CH,CH
        MOV  SI,1          ;в SI множитель
MET:    PUSH SI            ;сохраняем SI (множитель) в стеке
        MOV  SI,CX  ;в SI помещаем номер текущего символа
        MOV  AX,[BX+SI];в AX помещаем текущий символ
        XOR  AH,AH
        POP  SI            ;извлекаем множитель (SI)из стека
        SUB  AX,30H        ;получаем из символа (AX) цифру
        MUL  SI            ;умножаем цифру (AX)на множитель (SI)
        ADD  DI,AX ;складываем с результирующим числом
        MOV  AX,SI ;помещаем множитель (SI) в AX
        MOV  DX,10
        MUL  DX            ;увеличиваем множитель (AX) в 10 раз
        MOV  SI,AX ;перемещаем множитель (AX) назад в SI
        LOOP MET           ;переходим к предыдущему символу
```

5. Перевод чисел в различные системы счисления

В большинстве случаев перевод из одной системы счисления в другую осуществляется последовательным делением. При переводе из десятичной системы счисления в двоичную, восьмеричную и шестнадцатеричную алгоритм можно значительно упростить, заменив деление сдвигом.

Перевод из десятичной системы счисления в двоичную осуществляется последовательными сдвигами на один бит вправо. Таким образом, значение очередного

бита можно вычислить, проанализировав флаг переноса CF (если CF=1 то анализируемый бит был равен 1, и если CF=0, то анализируемый бит – 0).
Перевод из десятичной системы счисления в восьмеричную осуществляется последовательными сдвигами на три бита вправо. После очередного сдвига все биты кроме трех младших обнуляются (например, наложением маски командой AND). Таким образом, в регистре получается восьмеричная цифра, для получения ее символьного отображения к значению в регистре необходимо прибавить код нуля (30h).
Перевод из десятичной системы счисления в шестнадцатеричную осуществляется последовательными сдвигами на четыре бита вправо. После обнуления всех битов кроме четырех младших в регистре получается десятичный эквивалент шестнадцатеричной цифры (число от 0 до 15). Для его представления в шестнадцатеричной символьной форме необходимо организовать таблицу соответствия, которая в простейшем случае представляет собой следующую строку «0123456789ABCDEF». При перекодировании значение десятичного эквивалента используется как смещение в таблице относительно ее начала (перекодировка может осуществляться при помощи команды XLAT).

Задание:

- Разработать программу перевода чисел из десятичной системы счисления в двоичную, восьмеричную и шестнадцатеричную.
- Числа должны вводиться в десятичной системе счисления, а выводятся – в двоичной, восьмеричной и шестнадцатеричной.

ЛАБОРАТОРНАЯ РАБОТА №4

1. ТЕМА: Программирование обработчиков прерываний

2. ЦЕЛЬ РАБОТЫ:

Программная реализация различных способов переопределения векторов прерываний, написание обработчика прерывания.

3. ЗАДАНИЕ:

Разработать программу переопределения прерывания 05h (клавиша PrintScreen).

4. ЗАМЕЧАНИЯ:

- Перед загрузкой нового вектора прерывания необходимо сохранить старый вектор (функция 35h прерывания 21h).
- Новый обработчик прерывания должен быть FAR-процедурой.
- Для проверки, новая процедура обработки прерывания 05h должна выводить в динамик сигнал (прерывание 21H). В основной программе необходимо организовать большой цикл, например, выводящий на экран символы (прерывание 21h использовать нельзя, можно использовать, например, прерывание 10h). Таким образом, при нажатии на PrintScreen во время этого цикла компьютер должен издавать сигнал.

Прерывания и их переопределение. Иногда необходимо выполнить одну из набора специальных процедур, если в системе или в программе возникают определенные условия, например, нажата клавиша на клавиатуре. Действие, стимулирующее выполнение одной из таких процедур, называется прерыванием. Существует два общих класса прерываний: внутренние и внешние. Первые инициируются состоянием ЦП или командой, а вторые - сигналом, подаваемым от других компонентов системы.
Переход к процедуре прерывания осуществляется из любой программы, а после выполнения процедуры прерывания обязательно происходит возврат в прерванную программу. Перед обращением к процедуре прерывания должно быть сохранено

состояние всех регистров и флагов, используемых процедурой прерывания, а после окончания прерывания эти регистры должны быть восстановлены.
Последовательность прерывания состоит в следующем:

- текущее значение регистра *Flags* включается в стек;
- текущее значение регистра *CS* включается в стек;
- текущее значение регистра *IP* включается в стек;
- сбрасываются флаги *IF* и *TF*.

Новое содержимое *IP* и *CS* определяет начальный адрес выполняемой процедуры прерывания (обслуживание прерывания). Возврат в прерванную программу осуществляется командой, которая извлекает из стека содержимое для *IP*, *CS* и регистра флагов (обычно это команда *IRET*).
Адреса подпрограмм обслуживания прерываний (вектора прерываний) хранятся в таблице векторов прерываний. Таблица векторов прерываний располагается по адресу 0000:0000 и представляет собой массив из 256 элементов, каждый элемент которого занимает 4 байта и представляет собой начальный адрес процедуры обработки прерывания.
Иногда в программе возникает необходимость переопределения (перехвата) прерываний (например, выполнение дополнительных действий при нажатии определенной клавиши клавиатуры). Процесс перехвата прерываний состоит в следующем:

- подготавливается FAR-процедура – новый обработчик прерываний (должна заканчиваться командой *IRET*);
- сохраняется старый вектор прерывания (функция 35h прерывания 21h)
- адрес нового обработчика заносится в таблицу векторов прерываний (функция 25h прерывания 21h);
- в конце программы происходит восстановление первоначального обработчика прерываний.

Функция 35h
Вход: AH=35H
AL=номер прерывания (00H до 0ffH)
Выход: ES:BX=адрес обработчика прерывания
Описание: Возвращает значение вектора прерывания для INT (AL), то есть загружает в BX 0000:[AL*4], а в ES - 0000:[(AL*4)+2].
Функция 25h
Вход: AH=25H
AL=номер прерывания (00H до 0ffH)
DS:DX=вектор прерывания (адрес подпрограммы)
Выход: нет

Описание: Устанавливает значение элемента таблицы векторов прерываний для прерывания с номером AL равным DS:DX. Это равносильно записи 4-байтового адреса в 0000:(AL*4), но, в отличие от прямой записи, в момент записи прерывания будут заблокированы.
Ниже приведен фрагмент программы, иллюстрирующий установку нового вектора прерывания вместо обработчика PrintScreen (05h).

```
CODE SEGMENT
        ASSUME      CS:CODE,DS:DATA,SS:STACK
;Процедура – новый обработчик прерывания
PRNSCR      PROC FAR
            . . .
            IRET
PRNSCR      ENDP
```

```
START:
        XOR   AX,AX          ;Обычное начало для EXE-программы
        MOV   BX,DATA
        MOV   DS,BX
        . . .
;Установка нового вектора
        LEA   DX,CS:PRNSCR ;В DX – смещения нового обработчика
        PUSH  DS      ;Сохранение сегментного регистра DS
        PUSH  CS      ;Следующие две строки: загрузка регистра DS
        POP   DS      ;значением из CS через стек
        MOV   AH,25H         ;номер функции
        MOV   AL,5           ;номер прерывания
        INT   21H            ;установка нового прерывания
        POP   DS             ;восстановление DS
        . . .
CODE ENDS
```

ПРИЛОЖЕНИЕ III

Итоговый тест:

Вариант1 Архитектура ЭВМ -3й курс

1. **Машинная команда это**
   a) сигнал для выполнения той или иной машинной операции
   b) последовательностью операторов, отражающих решение задачи на каком – либо языке программирования
   c) способ кодирования состояний устройства
   d) способ указания источника операндов
2. **Какого вида архитектуры компьютера не бывает?**
   a) принстонская архитектура.
   b) схемная архитектура
   c) гарвардская архитектура.
   d) совмещенная архитектура
   .
3. **Представьте десятичное число 12 (decimal) в двоичной системе счисления (binary)**
   a) 1100
   b) 1011
   c) 00010010
   d) 00100001
4. **Представьте десятичное число 23 (decimal) в двоично-десятичном эквиваленте BCD (Binary codet decimal)**
   a) 10111
   b) 00110010
   c) 00100011
   d) 100011
5. **Представьте десятичное число 42 (decimal) в шестнадцатиричной системе счисления (hex)**
   a) A2
   b) 2B
   c) 1B
   d) 2A
6. **Одной из альтернативных единиц измерения производительности процессора (по отношению к времени выполнения) является**
   a) MIPS - (миллион команд в секунду).
   b) потребляемая мощность (в Ваттах)
   c) время выполнения одной команды (nanosek.)
   d) разрядность команды(бит)

7. *Дополните следующее определение:* **байт это восемь последовательно расположенных битов, пронумерованных от 0 до 7,**
   a) при этом нулевой бит является самым младшим значащим битом;
   b) при этом нулевой бит является самым старшим значащим битом
   c) при этом положение нулевого бита плавающее и зависит от системы счисления
   d) при этом положение нулевого бита плавающее и зависит от вида данных в байте

8. *Дополните следующее определение:* **•Слово —это последовательность из**
   a) двух байт, имеющих последовательные адреса.
   b) четырех байтов, имеющих последовательные адреса.
   c) двух байт, адреса которых формируются процессором произвольно.
   d) четырех байтов, адреса которых формируются процессором произвольно

9. Младшим байтом слова микропроцессора Intel , называется байт, содержащий
   a) 15-й бит

b) 7-й бит
c) нулевой бит
d) 31-й бит

10. Приведите важную особенность микропроцессоров Intel при формировании слова

a) *старший байт всегда хранится по меньшему адресу*
b) *младший байт всегда хранится по большему адресу*
c) *адреса старшего и младшего байта формируются процессором произвольно*
d) *младший байт всегда хранится по меньшему адресу.*

11. Адресом слова считается

a) *Адрес его старшего байта*
b) *Адрес его младшего байта*
c) *Адрес любого из байтов*

12. Двойное *слово* – это последовательность из

a) четырех байт, расположенных по последовательным адресам, с нумерацией бит, начиная с 0
b) двух байт, расположенных по последовательным адресам, с нумерацией бит начиная с 0
c) четырех байт, расположенных по произвольным адресам, с нумерацией бит начиная с 0.
d) восьми байт, расположенных по последовательным адресам, с нумерацией бит начиная с 0

13. Адресом двойного слова считается

a) Адрес его младшего слова
b) Адрес его старшего слова
c) Адрес любого из слов

14. *Учетверенное слово* — это последовательность из

a) четырех байт, расположенных по последовательным адресам
b) двух байт, расположенных по последовательным адресам,
c) восьми байт, расположенных по последовательным адресам.
d) Четырех байт, расположенных по произвольным адресам.

15. Какой из битов является знаковым при двоичном представлении чисел, в формате с фиксированной запятой в виде *целого числа со знаком*?

a) Младший бит (0)
b) Номер знакового бита формируется процессором произвольным образом
c) Бит с номером (3)
d) Старший бит (7, 15, или31)

16. Какому числу соответствует ноль (0) в знаковом бите при двоичном представлении чисел в формате с фиксированной запятой в виде ц*елого типа со знаком?*

a) Отрицательному числу
b) Положительному числу
c) Мнимому числу
d) Числу, представленному в комплексной форме

17. Представьте десятичное число 42 (dec) в слове длиной 16 бит в виде неупакованного десятичного (BSD) числа

a) 0000001000000100
b) 0000000001000010
c) 0000010000000010
d) 0100001000000000

18. Представьте десятичное число 42 (dec) в слове длиной 16 бит в виде упакованного десятичного (BSD) числа

a) 0000010000000010

b) 0000001000000100
c) 0100001000000000
d) 0000000001000010

19. Что представляет собой такой логический тип данных микропроцессора, как цепочка (байтовая строка)?

a) непрерывный набор байтов, слов или двойных слов максимальной длины до 4 Гбайт.

b) непрерывную последовательность бит, в которой каждый бит является независимым и может рассматриваться как отдельная переменная.

c) 32-разрядный логический адрес, представляющий собой относительное смещение в байтах от начала сегмента

d) 48-разрядный логический адрес, состоящий из двух частей: 16-разрядной сегментной части — селектора, и 32-разрядного смещения.

20. Представьте отрицательное десятичное число.-48,5(dec) в форме с плавающей запятой

a) $-0,0485*10^3$
b) $-4,85*10^1$
c) $-0,485*10^2$
d) **$--48,5*10^0$**

21. Представьте отрицательное десятичное число -12 (dec) в обратном (инверсном) коде

a) 1 1100
b) 0 0011
c) 0 1100
d) 1 0011

22. В какой из блоков Центрального процессора (ЦП) поступают операнды для выполнения операций над ними?

a) в арифметико-логическое устройство (АЛУ)
b) в устройство управления (УУ)
c) в оперативное запоминающее устройство (ОЗУ)
d) одновременно и в арифметико-логическое устройство (АЛУ) и в устройство управления (УУ)

23. В какой из блоков Центрального процессора (ЦП) поступают команды программы?

a) в арифметико-логическое устройство (АЛУ)
b) одновременно в арифметико-логическое устройство (АЛУ) и в устройство управления(УУ)
c) в устройство управления (УУ)
d) в оперативное запоминающее устройство (ОЗУ)

24. В каком блоке Центрального процессора (ЦП) хранятся микропрограммы команд?

a) в арифметико-логическом устройстве (АЛУ)
b) в двух блоках: и в арифметико-логическом устройстве (АЛУ) и в устройстве управления (УУ)
c) в оперативном запоминающем устройстве (ОЗУ)
d) в устройстве управления (УУ)

25. Какая из операций не входят в стандартный набор операций арифметико-логического устройства (АЛУ) Центрального процессора (ЦП)?

a) операция передачи из регистра в регистр РгА:=РгВ, РгВ:=РгА.
b) операция возведения в степень $РгА=РгА^п$
c) операция счета СчА:=А+1, СчВ:=В-1.
d) операция сложения, вычитания PrS:=A+B, PrS:=A-B

26. Установкой какого флага в регистре флагов (признаков) eflags/flags фиксируется факт нулевого результата при выполнении арифметических операций ?

a) Zf
b) ACf
c) S f
d) Of
E) Pf
F) Cf

27. Установкой какого флага в регистре флагов (признаков) eflags/flags фиксируется факт переполнения при выполнении арифметических операций ?

a) Zf
b) ACf
c) S f
d) Of
e) Pf
f) Cf

28. Установкой какого флага в регистре флагов (признаков) eflags/flags фиксируется знак результата при выполнении арифметических операций ?

a) Zf
b) ACf
c) S f
d) Of
E) Pf
F) Cf

29. Установкой какого флага в регистре флагов (признаков) eflags/flags фиксируется перенос результата при выполнении арифметических операций ?

a) Zf
b) ACf
c) S f
d) Of
E) Pf
F) Cf

30. Установкой какого флага в регистре флагов (признаков) eflags/flags фиксируется факт паритета результата при выполнении арифметических операций ?

a) Zf
b) ACf
c) S f
d) Of
E) Pf
F) Cf

31. Каким двоичным числом корректируется результат сложения двоично-десятичных чисел, если результат превышает цифру 9(dec) (1001(bin))?

a) 0110(bin)
b) 1001(bin)

c) 1011(bin)
d) 0101(bin)

32. Какой результат будет, если сложить два числа 8(dec) и 6 (dec) в двоично-десятичном коде с выполнением, при необходимости, десятичной коррекции?

a) 1 0011(bin)
b) 0 1110(bin)
c) 1 1110(bin)
d) 1 0100(bin)

33. При ведите результат выполнения логической операции И (AND) над логическими данными 0011 * 0010

a) =0010
b) =0111
c) =1100
d)=0101

34. Приведите результат выполнения логической операции M2 сложения по модулю 2 (XOR) над двоичными числами 0011(bin) (+) 0010(bin)

a) =0101
b) =0010
c) =0111
d) =1100