ИНФОРМАТИКА
И ИНФОРМАЦИОННЫЕ
ТЕХНОЛОГИИ
КОНСПЕКТ ЛЕКЦИЙ

ЛЕКЦИЯ № 1. Введение в информатику

1. Информатика. Информация. Представление и обработка информации

Информатика занимается формализованным представлением объектов и структур их взаимосвязей в различных областях науки, техники, производства. Для моделирования объектов и явлений используются различные формальные средства, например логические формулы, структуры данных, языки программирования и др.

В информатике такое фундаментальное понятие, как **информация** имеет различные значения:

1) формальное представление внешних форм информации;
2) абстрактное значение информации, ее внутреннее содержание, семантика;
3) отношение информации к реальному миру.

Но, как правило, под информацией понимают ее абстрактное значение — *семантику*. Интерпретируя представления информации, мы получим ее смысл, семантику. Поэтому, если мы хотим обмениваться информацией, нам необходимы согласованные представления, чтобы не нарушалась правильность интерпретации. Для этого интерпретацию представления информации отождествляют с некоторыми математическими структурами. В этом случае обработка информации может быть выполнена строгими математическими методами.

Одно из математических описаний информации — это представление ее в виде функции $y = f(x, t)$, где t — время, x — точка некоторого поля, в которой измеряется значение y. В зависимости от параметров функции x и t информацию можно классифицировать.

Если параметры — скалярные величины, принимающие непрерывный ряд значений, то полученная таким образом информация называется непрерывной (или *аналоговой)*. Если же параметрам придать некоторый шаг изменений, то информация называется

дискретной. Дискретная информация считается универсальной, так как для каждых конкретных параметров можно получить значение функции с заданной степенью точности.

Дискретную информацию обычно отождествляют с *цифровой* информацией, которая является частным случаем символьной информации алфавитного представления. **Алфавит** — конечный набор символов любой природы. Очень часто в информатике возникает ситуация, когда символы одного алфавита надо представить символами другого, т. е. провести операцию *кодирования*. Если количество символов кодируемого алфавита меньше количества символов кодирующего алфавита, то сама операция кодирования не является сложной, в противном случае необходимо использовать фиксированный набор символов кодирующего алфавита для однозначного правильного кодирования.

Как показала практика, наиболее простым алфавитом, позволяющим кодировать другие алфавиты, является двоичный, состоящий из двух символов, которые обозначаются, как правило, через 0 и 1. С помощью n символов двоичного алфавита можно закодировать 2n символов, а этого достаточно, чтобы закодировать любой алфавит.

Величина, которая может быть представлена символом двоичного алфавита, называется минимальной единицей информации или **битом**. Последовательность из 8 бит — **байт**. Алфавит, содержащий 256 различных 8-битных последовательностей, называется *байтовым*.

В качестве стандартного сегодня в информатике принят код, в котором каждый символ кодируется 1 байтом. Существуют и другие алфавиты.

2. Системы счисления

Под **системой счисления** подразумевается набор правил наименования и записи чисел. Различают позиционные и непозиционные системы счисления.

Система счисления называется *позиционной*, если значение цифры числа зависит от местоположения цифры в числе. В противном случае она называется *непозиционной*. Значение числа определяется по положению этих цифр в числе.

3. Представление чисел в ЭВМ

32-разрядные процессоры могут работать с оперативной памятью емкостью до 232-1, а адреса могут записываться в диапазоне 00000000 — FFFFFFFF. Однако в реальном режиме процессор работает с памятью до 220-1, а адреса попадают в диапазон 00000 — FFFFF. Байты памяти могут объединяться в поля как фиксированной, так и переменной длины. **Словом** называется поле фиксированной длины, состоящее из 2 байтов, **двойным словом** — поле из 4 байтов. Адреса полей бывают *четные* и *нечетные*, при этом для четных адресов операции выполняются быстрее.

Числа с фиксированной точкой в ЭВМ представляются как целые двоичные числа, и занимаемый ими объем может составлять 1, 2 или 4 байта.

Целые двоичные числа представляются в дополнительном коде, соответственно числа с фиксированной точкой представляются в дополнительном коде. При этом если число занимает 2 байта, то структура числа записывается по следующему правилу: старший разряд отводится под знак числа, а остальные — под двоичные цифры числа. Дополнительный код положительного числа равен самому числу, а дополнительный код отрицательного числа может быть получен по такой формуле: $x = 10n - |x|$, где n — разрядность числа.

В двоичной системе счисления дополнительный код получается путем инверсии разрядов, т. е., заменой единиц нулями и наоборот, и прибавлением единицы к младшему разряду.

Количество битов мантиссы определяет точность представления чисел, количество битов машинного порядка определяет диапазон представления чисел с плавающей точкой.

4. Формализованное понятие алгоритма

Алгоритм может существовать только тогда, когда в то же самое время существует некоторый математический объект. Формализованное понятие алгоритма связано с понятием рекурсивных функций, нормальных алгоритмов Маркова, машин Тьюринга.

В математике функция называется однозначной, если для любого набора аргументов существует закон, по которому опреде-

ляется единственное значение функции. В качестве такого закона может выступать алгоритм; в этом случае функция называется ***вычислимой***.

Рекурсивные функции — это подкласс вычислимых функций, а алгоритмы, определяющие вычисления, называются *сопутствующими алгоритмами* рекурсивных функций. Сначала фиксируются базовые рекурсивные функции, для которых сопутствующий алгоритм тривиален, однозначен; затем вводятся три правила — операторы *подстановки*, *рекурсии* и *минимизации*, при помощи которых на основе базовых функций получаются более сложные рекурсивные функции.

Базовыми функциями и их сопутствующими алгоритмами могут выступать:

1) функция n независимых переменных, тождественно равная нулю. Тогда, если знаком функции является φn, то независимо от количества аргументов значение функции следует положить равным нулю;

2) тождественная функция *n* независимых переменных вида Ψni. Тогда, если знаком функции является Ψni, то значением функции следует взять значение i-го аргумента, считая слева направо;

3) λ -функция одного независимого аргумента. Тогда, если знаком функции является λ, то значением функции следует взять значение, следующее за значением аргумента.

Разные ученые предлагали свои подходы к формализованному представлению алгоритма. Например, американский ученый Черч предположил, что класс вычислимых функций исчерпывается рекурсивными функциями и, как следствие, каким бы ни был алгоритм, перерабатывающий один набор целых неотрицательных чисел в другой, найдется алгоритм, сопутствующий рекурсивной функции, эквивалентный данному. Следовательно, если для решения некоторой поставленной задачи нельзя построить рекурсивную функцию, то и не существует алгоритма для ее решения. Другой ученый, Тьюринг, разработал виртуальную ЭВМ, которая перерабатывала входную последовательность символов в выходную. В связи с этим им был выдвинут тезис, что любая вычислимая функция вычислима по Тьюрингу.

ЛЕКЦИЯ № 2. Язык Pascal

1. Введение в язык Pascal

Основные символы языка — буквы, цифры и специальные символы — составляют его алфавит. Язык Pascal включает следующий набор основных символов:

1) 26 латинских строчных и 26 латинских прописных букв:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

a b c d e f g h i j k l m n o p q r s t u v w x y z;

2) _ (знак подчеркивания) ;

3) 10 цифр: 0 1 2 3 4 5 6 7 8 9;

4) знаки операций:

\+ — × / = <> < > <= >= := @ ;

5) ограничители:

. , ' ( ) [ ] (. .) { } (* *) .. : ;

6) спецификаторы: ^ # $;

7) служебные (зарезервированные) слова:

ABSOLUTE, ASSEMBLER, AND, ARRAY, ASM, BEGIN, CASE, CONST, CONSTRUCTOR, DESTRUCTOR, DIV, DO, DOWNTO, ELSE, END, EXPORT, EXTERNAL, FAR, FILE, FOR, FORWARD, FUNCTION, GOTO, IF, IMPLEMENTATION, IN, INDEX, INHERITED, INLINE, INTERFACE, INTERRUPT, LABEL, LIBRARY, MOD, NAME, NIL, NEAR, NOT, OBJECT, OF, OR, PACKED, PRIVATE, PROCEDURE, PROGRAM, PUBLIC, RECORD, REPEAT, RESIDENT, SET, SHL, SHR, STRING, THEN, TO, TYPE, UNIT, UNTIL, USES, VAR, VIRTUAL, WHILE, WITH, XOR.

Кроме перечисленных, в набор основных символов входит пробел. Пробелы нельзя использовать внутри сдвоенных символов и зарезервированных слов.

Концепция типа для данных

В математике принято классифицировать переменные в соответствии с некоторыми важными характеристиками. Произво-

дится строгое разграничение между вещественными, комплексными и логическими переменными, между переменными, представляющими отдельные значения и множество значений, и т. д. При обработке данных на ЭВМ такая классификация еще более важна. В любом алгоритмическом языке каждая константа, переменная, выражение или функция бывают определенного типа.

В языке Pascal существует правило: тип явно задается в описании переменной или функции, которое предшествует их использованию. Концепция типа языка Pascal имеет следующие основные свойства:

1) любой тип данных определяет множество значений, к которому принадлежит константа, которые может принимать переменная или выражение либо вырабатывать операция или функция;
2) тип значения, задаваемого константой, переменной или выражением, можно определить по их виду или описанию;
3) каждая операция или функция требуют аргументов фиксированного типа и выдают результат фиксированного типа.

Отсюда следует, что транслятор может использовать информацию о типах для проверки вычислимости и правильности различных конструкций.

Тип определяет:

1) возможные значения переменных, констант, функций, выражений, принадлежащих к данному типу;
2) внутреннюю форму представления данных в ЭВМ;
3) операции и функции, которые могут выполняться над величинами, принадлежащими к данному типу.

Следует заметить, что обязательное описание типа приводит к избыточности в тексте программ, но такая избыточность является важным вспомогательным средством разработки программ и рассматривается как необходимое свойство современных алгоритмических языков высокого уровня.

В языке Pascal существуют скалярные и структурированные типы данных. К скалярным типам относятся стандартные типы и типы, определяемые пользователем. Стандартные типы включают целые, действительные, символьный, логические и адресный типы.

Целые типы определяют константы, переменные и функции, значения которых реализуются множеством целых чисел, допустимых в данной ЭВМ.

Тип	Диапазон значений	Требуемая память (байт)
Byte	0...255	1
Word	0...65535	2
Shortint	–128...127	1
Integer	–32768...32767	2
Longint	– 2147483648...2147483647	4

Действительные типы определяет те данные, которые реализуются подмножеством действительных чисел, допустимых в данной ЭВМ.

Тип	Диапазон значений	Количество цифр мантиссы	Требуемая память (байт)
Real	2.9e – 39...1.7e + 38	11–12	6
Single	1.5e – 45...3.4e + 38	7–8	4
Double	5.0e – 324...1.7e + 308	15–16	8
Extended	3.4e – 4932...1.1e + 4932	19–20	10
Comp	–9.2e + 18...9.2e + 18	19–20	2

Типы, определяемые пользователем, — перечисляемый и интервальный. Структурированные типы имеют четыре разновидности: массивы, множества, записи и файлы.

Кроме перечисленных, Pascal включает еще два типа — процедурный и объектный.

Выражение языка состоит из констант, переменных, указателей функций, знаков операций и скобок. Выражение задает правило вычисления некоторого значения. Порядок вычисления определяется старшинством (приоритетом) содержащихся в нем операций. В языке Pascal принят следующий приоритет операций:

1) вычисления в круглых скобках;
2) вычисления значений функций;
3) унарные операции;
4) операции *, /, div, mod, and;
5) операции +, — , or, xor;
6) операции отношения =, <>, <, >, <=, >=.

Выражения входят в состав многих операторов языка Pascal, — также могут быть аргументами встроенных функций.

2. Стандартные процедуры и функции

Арифметические функции

1. Function Abs(X);

Возвращает абсолютное значение параметра.

X — выражение вещественного или целочисленного типа.

2. Function ArcTan(X: Extended): Extended;

Возвращает арктангенс аргумента.

X — выражение вещественного или целочисленного типа.

3. Function Exp(X: Real): Real;

Возвращает экспоненту.

X — выражение вещественного или целочисленного типа.

4. Function Frac(X: Real): Real;

Возвращает дробную часть аргумента.

X — выражение вещественного типа. Результат — дробная часть X, т. е.

Frac (X) = X – Int (X).

5. Function Int(X: Real): Real;

Возвращает целочисленную часть аргумента.

X — выражение вещественного типа. Результат — целочисленная часть X, т. е. X, округленный к нулю.

6. Function Ln(X: Real): Real;

Возвращает натуральный логарифм (Ln e = 1) выражения X вещественного типа.

7. Function Pi: Extended;

Возвращает значение Pi, которое определено как 3.1415926535.

8. Function Sin(X: Extended): Extended;

Возвращает синус аргумента.

X — выражение вещественного типа. Sin возвращает синус угла X в радианах.

9. Function Sqr(X: Extended): Extended;

Возвращает квадрат аргумента.

X — выражение с плавающей запятой. Результат того же самого типа, что и X.

10. Function Sqrt(X: Extended): Extended;

Возвращает квадратный корень аргумента.

X — выражение с плавающей запятой. Результат — квадратный корень X.

Процедуры и функции преобразования величин

1. Procedure Str(X [: Width [: Decimals]]; var S);

Преобразовывает число X в строковое представление согласно Width и параметрам форматирования Decimals. X — выражение вещественного или целого типа. Width и Decimals — выражения целого типа. S — переменная типа String или символьный массив с нулевым окончанием, если допускается расширенный синтаксис.

2. Function Chr(X: Byte): Char;

Возвращает символ с порядковым номером X в ASCII-таблице.

3. Function High(X);

Возвращает наибольшее значение в диапазоне параметра.

4. Function Low(X);

Возвращает наименьшее значение в диапазоне параметра.

5. Function Ord(X): LongInt;

Возвращает порядковое значение выражения перечислимого типа. X — выражение перечислимого типа.

6. Function Round(X: Extended): LongInt;

Округляет значение вещественного типа до целого. X — выражение вещественного типа. Round возвращает значение Longint, которое является значением X, округленным до ближайшего целого числа. Если X находится точно посередине между двумя целыми числами, возвращается число с наибольшей абсолютной величиной. Если округленное значение X выходит за диапазон LongInt, генерируется ошибка времени выполнения программы, которую вы можете обработать с использованием исключительной ситуации EInvalidOp.

7. Function Trunc(X: Extended): LongInt;

Усекает значение вещественного типа до целого. Если округленное значение X выходит за диапазон LongInt, генерируется ошибка времени выполнения программы, которую вы можете обработать с использованием исключительной ситуации EInvalidOp.

8. Procedure Val(S; var V; var Code: Integer);

Преобразовывает число из строкового значения S в числовое представление V. S — выражение строкового типа — последовательность символов, которая формирует целое или вещественное число. Если выражение S недопустимо, индекс неверного символа сохраняется в переменной Code. В противном случае Code устанавливается в нуль.

Процедуры и функции работы с порядковыми величинами

1. Procedure Dec(var X [; N: LongInt]);

Вычитает единицу или N из переменной X. Dec(X) соответствует X := X — 1, и Dec(X, N) соответствует X := X – N. X — переменная перечислимого типа или типа PChar, если допускается расширенный синтаксис, и N — выражение целочисленного типа. Процедура Dec генерирует оптимальный код и особенно полезна в длительных циклах.

2. Procedure Inc(var X [; N: LongInt]);

Прибавляет единицу или N к переменной X. X — переменная перечислимого типа или типа PChar, если допускается расширенный синтаксис, и N — выражение целочисленного типа. Inc (X) соответствует инструкции X := X + 1, и Inc (X, N) соответствует инструкции X := X + N. Процедура Inc генерирует оптимальный код и особенно полезна в длительных циклах.

3. Function Odd(X: LongInt): Boolean;

Возвращает True, если X — нечетное число, и False — в противном случае.

4. Function Pred(X);

Возвращает предыдущее значение параметра. X — выражение перечислимого типа. Результат того же самого типа.

5. Function Succ(X);

Возвращает следующее значение параметра. X — выражение перечислимого типа. Результат того же самого типа.

3. Операторы языка Pascal

Условный оператор

Формат полного условного оператора определяется следующим образом: If B then S1 else S2; где B — условие разветвления (принятия решения), логическое выражение или отношение; S1, S2 — один выполняемый оператор, простой или составной.

При выполнении условного оператора сначала вычисляется выражение B, затем анализируется его результат: если B — истинно, то выполняется оператор S1 — ветвь then, а оператор S2 пропускается; если B — ложно, то выполняется оператор S2 — ветвь else, а оператор S1 — пропускается.

Также существует сокращенная форма условного оператора. Она записывается в виде: *If B then S*.

Оператор выбора

Структура оператора имеет следующий вид:

case S of
*c*1*: insruction*1*;*
*c2: insruction*2*;*
...
cn: insructionN;
else instruction
end;

где *S* — выражение порядкового типа, значение которого вычисляется;
*c*1, *c*2,..., *cn* — константы порядкового типа, с которыми сравниваются выражения
S; instruction1,..., instructionN — операторы, из которых выполняется тот, с константой которого совпадает значение выражения *S*;
instruction — оператор, который выполняется, если значение выражения *S* не совпадает ни с одной из констант c1, c2, ... cn.

Данный оператор является обобщением условного оператора If для произвольного числа альтернатив. Существует сокращенная форма оператора, при которой ветвь *else* отсутствует.

Оператор цикла с параметром

Операторы цикла с параметром, которые начинаются со слова *for*, вызывают повторяющееся выполнение оператора, который может быть составным оператором, пока управляющей переменной присваивается возрастающая последовательность значений. Общий вид оператора *for*:

for <счетчик цикла> := <начальное значение> to <конечное значение> do <оператор>;

Когда начинает выполняться оператор *for*, начальное и конечное значения определяются один раз, и эти значения сохраняются на протяжении всего выполнения оператора *for*. Оператор, который содержится в теле оператора *for*, выполняется один раз для каждого значения в диапазоне между начальным и конечным значением. Счетчик цикла всегда инициализируется начальным значением. Когда работает оператор *for*, значение счетчика цикла увеличивается при каждом повторении на единицу. Если начальное значение превышает конечное значение, то содержащийся в теле оператора for оператор не выполняется. Когда в операторе цикла используется ключевое слово *downto*, значение управляющей переменной уменьшается при каждом повторении на единицу. Если начальное значение в таком операторе меньше, чем конечное значение, то содержащийся в теле оператора цикла оператор не выполняется.

Если оператор, содержащийся в теле оператора *for*, изменяет значение счетчика цикла, то это является ошибкой. После выполнения оператора *for* значение управляющей переменной становится неопределенным, если только выполнение оператора *for* не было прервано с помощью оператора перехода.

Оператор цикла с предусловием

Оператор цикла с предусловием (начинающийся с ключевого слова while) содержит в себе выражение, которое управляет повторным выполнением оператора (который может быть составным оператором). Форма цикла:

While B do S;

где *B* — логическое условие, истинность которого проверяется (оно является условием завершения цикла)$

S — тело цикла — один оператор.

Выражение, с помощью которого осуществляется управление повторением оператора, должно иметь логический тип. Вычисление его производится до того, как внутренний оператор будет выполнен. Внутренний оператор выполняется повторно до тех пор, пока выражение принимает значение True. Если выражение с самого начала принимает значение False, то оператор, содержащийся внутри оператора цикла с предусловием, не выполняется.

Оператор цикла с постусловием

В операторе цикла с постусловием (начинающимся со слова *repeat*) выражение, которое управляет повторным выполнением последовательности операторов, содержится внутри оператора *repeat*. Форма цикла:

repeat S until B;

где *B* — логическое условие, истинность которого проверяется (оно является условием завершения цикла);
S — один или более операторов тела цикла.

Результат выражения должен быть логического типа. Операторы, заключенные между ключевыми словами *repeat* и *until*, выполняются последовательно до тех пор, пока результат выражения не примет значение True. Последовательность операторов выполнится, по крайней мере, один раз, поскольку вычисление выражения производится после каждого выполнения последовательности операторов.

ЛЕКЦИЯ № 3. Процедуры и функции

1. Понятие вспомогательного алгоритма

Алгоритм решения задачи проектируется путем декомпозиции всей задачи в отдельные подзадачи. Обычно подзадачи реализуются в виде подпрограмм.

Подпрограмма — это некоторый вспомогательный алгоритм, многократно использующийся в основном алгоритме с различными значениями некоторых входящих величин, называемых параметрами.

Подпрограмма в языках программирования — это последовательность операторов, которые определены и записаны только в одном месте программы, однако их можно вызвать для выполнения из одной или нескольких точек программы. Каждая подпрограмма определяется уникальным именем.

В языке Pascal существуют два типа подпрограмм — процедуры и функции. Процедура и функция — это именованная последовательность описаний и операторов. При использовании процедур или функций программа должна содержать текст процедуры или функции и обращение к процедуре или функции. Параметры, указанные в описании, называются формальными, указанные в обращении подпрограммы — фактическими. Все формальные параметры можно разбить на следующие категории:

1) параметры-переменные;
2) параметры-константы;
3) параметры-значения;
4) параметры-процедуры и параметры-функции, т. е. параметры процедурного типа;
5) нетипизированные параметры-переменные.

Тексты процедур и функций помещаются в раздел описаний процедур и функций.

Передача имен процедур и функций в качестве параметров

Во многих задачах, особенно в задачах вычислительной математики, необходимо передавать имена процедур и функций в ка-

честве параметров. Для этого в TURBO PASCAL введен новый тип данных — процедурный, или функциональный, в зависимости от того, что описывается. (Описание процедурных и функциональных типов приводится в разделе описания типов.)

Функциональный и процедурный тип определяется как заголовок процедуры и функции со списком формальных параметров, но без имени. Можно определить функциональный, или процедурный тип без параметров, например:

```
type
Proc = Procedure;
```

После объявления процедурного, или функционального, типа его можно использовать для описания формальных параметров — имен процедур и функций. Кроме того, необходимо написать те реальные процедуры или функции, имена которых будут передаваться как фактические параметры.

2. Процедуры в Pascal

Каждое описание процедуры содержит заголовок, за которым следует программный блок. Общий вид заголовка процедуры следующий:

Procedure <имя> [(*<список формальных параметров>*)]*;*

Процедура активизируется с помощью оператора процедуры, в котором содержатся имя процедуры и необходимые параметры. Операторы, которые должны выполняться при запуске процедуры, содержатся в операторной части модуля процедуры. Если в содержащемся в процедуре операторе внутри модуля процедуры используется идентификатор процедуры, то процедура будет выполняться рекурсивно, т. е. будет при выполнении обращаться сама к себе.

3. Функции в Pascal

Описание функции определяет часть программы, в которой вычисляется и возвращается значение. Общий вид заголовка функции следующий:

Function <имя> [(*<список формальных параметров>*)]*: <тип возвращаемого результата>;*

Функция активизируется при ее вызове. При вызове функции указываются идентификатор функции и какие-либо параметры, необходимые для ее вычисления. Вызов функции может включаться в выражения в качестве операнда. Когда выражение вычисляется, функция выполняется и значением операнда становится значение, возвращаемое функцией.

В операторной части блока функции задаются операторы, которые должны выполняться при активизации функции. В модуле должен содержаться, по крайней мере, один оператор присваивания, в котором идентификатору функции присваивается значение. Результатом функции является последнее присвоенное значение. Если такой оператор присваивания отсутствует или он не был выполнен, то значение, возвращаемое функцией, не определено.

Если идентификатор функции используется при вызове функции внутри модуля, то функция выполняется рекурсивно.

4. Опережающие описания и подключение подпрограмм. Директива

В программе может содержаться несколько подпрограмм, т. е. структура программы может быть усложнена. Однако эти подпрограммы могут располагаться на одном уровне вложенности, поэтому сначала должно идти описание подпрограммы, а затем обращение к ней, если только не используется специальное опережающее описание.

Описание процедуры, содержащее вместо блока операторов директиву *forward*, называется опережающим описанием. В каком-либо месте после этого описания с помощью определяющего описания процедура должна определяться. Определяющее описание — это описание, в котором используется тот же идентификатор процедуры, но опущен список формальных параметров, и в которое включен блок операторов. Описание *forward* и определяющее описание должны присутствовать в одной и той же части описания процедуры и функции. Между ними могут описываться другие процедуры и функции, которые могут обращаться к процедуре с опережающим описанием. Таким образом, возможна взаимная рекурсия.

Опережающее описание и определяющее описание представляют собой полное описание процедуры. Процедура считается описанной с помощью опережающего описания.

Если в программе будет содержаться довольно много подпрограмм, то программа перестанет быть наглядной, в ней будет тяжело ориентироваться. Во избежание этого некоторые подпрограммы хранят в виде исходных файлов на диске, а при необходимости они подключаются к основной программе на этапе компиляции при помощи *директивы компиляции*.

Директива — это специальный комментарий, который может быть размещен в любом месте программы, там, где может находиться и обычный комментарий. Однако они различаются тем, что у директивы имеется специальная форма записи: сразу после закрывающей скобки без пробела записывается знак $, а затем, опять же без пробела, указывается директива.

Пример

1) {$E+} — эмулировать математический сопроцессор;
2) {$F+} — формировать дальний тип вызова процедур и функций;
3) {$N+} — использовать математический сопроцессор;
4) {$R+} — проверять выход за границы диапазонов.

Некоторые ключи компиляции могут содержать параметр, например:

{$I имя файла} — включить в текст компилируемой программы названный файл.

ЛЕКЦИЯ № 4. Подпрограммы

1. Параметры подпрограмм

В описании процедуры или функции задается список формальных параметров. Каждый параметр, описанный в списке формальных параметров, является локальным по отношению к описываемой процедуре или функции, и в модуле, связанном с данной процедурой или функцией, на него можно ссылаться по его идентификатору.

Существуют три типа параметров: значение, переменная и нетипизированная переменная. Они характеризуются следующим.

1. Группа параметров без предшествующего ключевого слова является списком параметров-значений.

2. Группа параметров, перед которыми следует ключевое слово const и за которыми следует тип, является списком параметров-констант.

3. Группа параметров, перед которыми стоит ключевое слово *var* и за которыми следует тип, является списком нетипизированных параметров-переменных.

4. Группа параметров, перед которыми стоит ключевое слово *var* или const, за которыми не следует тип, является списком нетипизированных параметров-переменных.

2. Типы параметров подпрограмм

Параметры-значения

Формальный параметр-значение обрабатывается как локальная по отношению к процедуре или функции переменная, за исключением того, что он получает свое начальное значение из соответствующего фактического параметра при активизации процедуры или функции. Изменения, которые претерпевает формальный параметр-значение, не влияют на значение фактического параметра. Соответствующее фактическое значение параметра-значения должно быть выражением, и его значение не должно иметь

файловый тип или какой-либо структурный тип, содержащий в себе файловый тип.

Фактический параметр должен иметь тип, совместимый по присваиванию с типом формального параметра-значения. Если параметр имеет строковый тип, то формальный параметр будет иметь атрибут размера, равный 255.

Параметры-константы

Формальные параметры-константы работают аналогично локальной переменной, доступной только по чтению, которая получает свое значение при активизации процедуры или функции от соответствующего фактического параметра. Присваивания формальному параметру-константе не допускаются. Формальный параметр-константа также не может передаваться в качестве фактического параметра другой процедуре или функции. Параметр-константа, соответствующий фактическому параметру в операторе процедуры или функции, должен подчиняться тем же правилам, что и фактическое значение параметра.

В тех случаях, когда формальный параметр не изменяет при выполнении процедуры или функции своего значения, вместо параметра-значения следует использовать параметр-константу. Параметры-константы позволяют при реализации процедуры или функции защититься от случайных присваиваний формальному параметру. Кроме того, для параметров структурного и строкового типа компилятор при использовании вместо параметров-значений параметров-констант может генерировать более эффективный код.

Параметры-переменные

Параметр-переменная используется, когда значение должно передаваться из процедуры или функции вызывающей программе. Соответствующий фактический параметр в операторе вызова процедуры или функции должен быть ссылкой на переменную. При активизации процедуры или функции формальный параметр-переменная замещается фактической переменной, любые изменения в значении формального параметра-переменной отражаются на фактическом параметре.

Внутри процедуры или функции любая ссылка на формальный параметр-переменную приводит к доступу к самому фактическому параметру. Тип фактического параметра должен совпадать с типом

формального параметра-переменной, но это ограничение можно обойти с помощью нетипизированного параметра-переменной).

Нетипизированные параметры

Когда формальный параметр является нетипизированным параметром-переменной, то соответствующий фактический параметр может представлять собой любую ссылку на переменную или константу независимо от ее типа. Нетипизированный параметр, описанный с ключевым словом var, может модифицироваться, а нетипизированный параметр, описанный с ключевым словом const, доступен только по чтению.

В процедуре или функции у нетипизированного параметра-переменной тип отсутствует, т. е. он несовместим с переменными всех типов, пока ему не будет присвоен определенный тип с помощью присваивания типа переменной.

Хотя нетипизированные параметры дают большую гибкость, их использование сопряжено с некоторым риском. Компилятор не может проверить допустимость операций с нетипизированными переменными.

Процедурные переменные

После определения процедурного типа появляется возможность описывать переменные этого типа. Такие переменные называют процедурными переменными. Как и целая переменная, которой можно присвоить значение целого типа, процедурной переменной можно присвоить значение процедурного типа. Таким значением может быть, конечно, другая процедурная переменная, но оно может также представлять собой идентификатор процедуры или функции. В таком контексте описания процедуры или функции можно рассматривать как описание особого рода константы, значением которой является процедура или функция.

Как и при любом другом присваивании, значения переменной в левой и в правой части должны быть совместимы по присваиванию. Процедурные типы, чтобы они были совместимы по присваиванию, должны иметь одно и то же число параметров, а параметры на соответствующих позициях должны быть одинакового типа. Имена параметров в описании процедурного типа никакого действия не вызывают.

Кроме того, для обеспечения совместимости по присваиванию процедура или функция, если ее нужно присвоить процедурной переменной, должна удовлетворять следующим требованиям:

1) это не должна быть стандартная процедура или функция;
2) такая процедура или функция не может быть вложенной;
3) такая процедура не должна быть процедурой типа *inline*;
4) она не должна быть процедурой прерывания (*interrupt*).

Стандартными процедурами и функциями считаются процедуры и функции, описанные в модуле System, такие как *Writeln, Readln, Chr, Ord*. Вложенные процедуры и функции с процедурными переменными использовать нельзя. Процедура или функция считается вложенной, когда она описывается внутри другой процедуры или функции.

Использование процедурных типов не ограничивается просто процедурными переменными. Как и любой другой тип, процедурный тип может участвовать в описании структурного типа.

Когда процедурной переменной присваивается значение процедуры, то на физическом уровне происходит следующее: адрес процедуры сохраняется в переменной. Фактически процедурная переменная весьма напоминает переменную-указатель, только вместо ссылки на данные она указывает на процедуру или функцию. Как и указатель, процедурная переменная занимает 4 байта (два слова), в которых содержится адрес памяти. В первом слове хранится смещение, во втором — сегмент.

Параметры процедурного типа

Поскольку процедурные типы допускается использовать в любом контексте, то можно описывать процедуры или функции, которые воспринимают процедуры и функции в качестве параметров. Параметры процедурного типа особенно полезны в том случае, когда над множеством процедур или функций нужно выполнить какие-то общие действия.

Если процедура или функция должны передаваться в качестве параметра, они должны удовлетворять тем же правилам совместимости типа, что и при присваивании. То есть, такие процедуры или функции должны компилироваться с директивой *far*, они не могут быть встроенными функциями, не могут быть вложенными и не могут описываться с атрибутами *inline* или *interrupt*.

ЛЕКЦИЯ № 5. Строковый тип данных

1. Строковый тип в Pascal

Последовательность символов определенной длины называется строкой. Переменные строкового типа определяются путем указания имени переменной, зарезервированного слова *string*, и возможно, но не обязательно указания максимального размера, т. е. длины строки, в квадратных скобках. Если не задавать максимальный размер строки, то по умолчанию он будет равен 255, т. е. строка будет состоять из 255 символов.

К каждому элементу строки можно обратиться по его номеру. Однако ввод и вывод строк осуществляются целиком, а не поэлементно, как это происходит в массивах. Число введенных символов не должно превышать указанного в максимальном размере строки, так если такое превышение будет иметь место, то «лишние» символы будут проигнорированы.

2. Процедуры и функции для переменных строкового типа

1. Function Copy(S: String; Index, Count: Integer): String;

Возвращает подстроку строки. S — выражение типа String. Index и Count — выражения целого типа. Функция возвращает строку, содержащую Count символов, начинающихся с позиции Index. Если Index больше, чем длина S, функция возвращает пустую строку.

2. Procedure Delete(var S: String; Index, Count:Integer);

Удаляет подстроку символов длиной Count из строки S, начиная с позиции Index. S — переменная типа String. Index и Count — выражения целого типа. Если Index больше, чем длина S, символы не удаляются.

3. Procedure Insert(Source: String; var S: String; Index: Integer);

Объединяет подстроку в строку, начиная с определенной позиции. Source — выражение типа String. S — переменная типа String любой длины. Index — выражение целочисленного типа. Insert вставляет Source в S, начиная с позиции S[Index].

4. Function Length(S: String): Integer;

Возвращает число символов, фактически используемое в строке S. Обратите внимание: при использовании строк с нуль-окончанием, число символов не обязательно равно числу байтов.

5. Function Pos(Substr: String; S: String): Integer;

Ищет подстроку в строке. Pos ищет Substr внутри S и возвращает целочисленное значение, которое является индексом первого символа Substr внутри S. Если Substr не найден, Pos возвращает нуль.

3. Записи

Запись представляет собой совокупность ограниченного чис-ла логически связанных компонент, принадлежащих к разным типам. Компоненты записи называются полями, каждое из которых определяется именем. Поле записи содержит имя поля, вслед за которым через двоеточие указывается тип этого поля. Поля записи могут относиться к любому типу, допустимому в языке Pascal, за исключением файлового типа.

Описание записи в языке Pascal осуществляется с помощью служебного слова RECORD, вслед за которым описываются компоненты записи. Завершается описание записи служебным словом END.

Например, записная книжка содержит фамилии, инициалы и номера телефона, поэтому отдельную строку в записной книжке удобно представить в виде следующей записи:

```
type Row = Record
 FIO: String[20];
 TEL: String[7];
end;
var str: Row;
```

Описание записей возможно и без использования имени типа, например:

```
var str : Record
 FIO : String[20];
 TEL : String[7];
end;
```

Обращение к записи в целом допускается только в операторах присваивания, где слева и справа от знака присваивания используются имена записей одинакового типа. Во всех остальных случаях оперируют отдельными полями записей. Чтобы обратиться к отдельной компоненте записи, необходимо задать имя записи и через точку указать имя нужного поля. Такое имя называется составным. Компонентой записи может быть также запись, в таком случае составное имя будет содержать не два, а большее количество имен.

Обращение к компонентам записей можно упростить, если воспользоваться оператором присоединения with. Он позволяет заменить составные имена, характеризующие каждое поле, просто на имена полей, а имя записи определить в операторе присоединения.

Иногда содержимое отдельной записи зависит от значения одного из ее полей. В языке Pascal допускается описание записи, состоящей из общей и вариантной частей. Вариантная часть задается с помощью конструкции *case P of*, где P — имя поля из общей части записи. Возможные значения, принимаемые этим полем, перечисляются так же, как и в операторе варианта. Однако вместо указания выполняемого действия, как это делается в операторе варианта, указываются поля варианта, заключенные в круглые скобки. Описание вариантной части завершается служебным словом end. Тип поля P можно указать в заголовке вариантной части. Инициализация записей осуществляется с помощью типизированных констант.

4. Множества

Понятие множества в языке Pascal основывается на математическом представлении о множествах: это ограниченная совокупность различных элементов. Для построения конкретного множественного типа используется перечисляемый или интервальный тип данных. Тип элементов, составляющих множество, называется базовым типом.

Множественный тип описывается с помощью служебных слов Set of, например:

type M = Set of B;

Здесь M — множественный тип, B — базовый тип.

Принадлежность переменных к множественному типу может быть определена прямо в разделе описания переменных.

Константы множественного типа записываются в виде заключенной в квадратные скобки последовательности элементов или интервалов базового типа, разделенных запятыми. Константа вида [ ] означает пустое подмножество.

Множество включает в себя набор элементов базового типа, все подмножества данного множества, а также пустое подмножество. Если базовый тип, на котором строится множество, имеет K элементов, то число подмножеств, входящих в это множество, равно 2 в степени K. Порядок перечисления элементов базового типа в константах безразличен. Значение переменной множественного типа может быть задано конструкцией вида [T], где T — переменная базового типа.

К переменным и константам множественного типа применимы операции присваивания (:=), объединения (+), пересечения (*) и вычитания (–). Результат выполнения этих операций есть величина множественного типа:

1) ['A','B'] + ['A','D'] даст ['A','B','D'];
2) ['A'] * ['A','B','C'] даст ['A'];
3) ['A','B','C'] — ['A','B'] даст ['C'].

К множественным величинам применимы операции: тождественность (=), нетождественность (<>), содержится в (<=), содержит (>=). Результат выполнения этих операций имеет логический тип:

1) ['A','B'] = ['A','C'] даст FALSE ;
2) ['A','B'] <> ['A','C'] даст TRUE;
3) ['B'] <= ['B','C'] даст TRUE;
4) ['C','D'] >= ['A'] даст FALSE.

Кроме этих операций, для работы с величинами множественного типа используется операция *in*, проверяющая принадлежность элемента базового типа, стоящего слева от знака операции, множеству, стоящему справа от знака операции. Результат выполнения этой операции — булевский. Операция проверки принадлежности элемента множеству часто используется вместо операций отношения.

При использовании в программах данных множественного типа выполнение операций происходит над битовыми строками данных. Каждому значению множественного типа в памяти ЭВМ соответствует один двоичный разряд.

Величины множественного типа не могут быть элементами списка ввода-вывода. В каждой конкретной реализации транслятора с языка Pascal количество элементов базового типа, на котором строится множество, ограниченно.

Инициализация величин множественного типа производится с помощью типизированных констант.

Приведем некоторые процедуры для работы с множествами.

1. Procedure Exclude(var S: Set of T; I:T);

Удаляет элемент I из множества S. S — переменная типа «множество», и I — выражение типа, совместимого с исходным типом S. Конструкция Exclude(S, I) соответствует S: = S – [I], но генерирует более эффективный код.

2. Procedure Include(var S: Set of T; I:T);

Добавляет элемент I к множеству S. S — переменная типа «множество», и I — выражение типа, совместимого с типом S. Конструкция Include(S, I) соответствует S: = S + [I], но генерирует более эффективный код.

ЛЕКЦИЯ № 6. Файлы

1. Файлы. Операции с файлами

Введение файлового типа в язык Pascal вызвано необходимостью обеспечить возможность работы с периферийными (внешними) устройствами ЭВМ, предназначенными для ввода, вывода и хранения данных.

Файловый тип данных (или файл) определяет упорядоченную совокупность произвольного числа однотипных компонент. Общее свойство массива, множества и записи заключается в том, что количество их компонент определено на этапе написания программы, тогда как количество компонент файла в тексте программы не определяется и может быть произвольным.

При работе с файлами выполняются операции ввода-вывода. Операция ввода означает перепись данных с внешнего устройства (из входного файла) в основную память ЭВМ, операция вывода — это пересылка данных из основной памяти на внешнее устройство (в выходной файл). Файлы на внешних устройствах часто называют физическими файлами. Их имена определяются операционной системой.

В программах на языке Pascal имена файлов задаются с помощью строк. Для работы с файлами в программе необходимо определить файловую переменную. Pascal поддерживает три файловых типа: текстовые файлы, компонентные файлы, бестиповые файлы.

Файловые переменные, которые описаны в программе, называют логическими файлами. Все основные процедуры и функции, обеспечивающие ввод-вывод данных, работают только с логическими файлами. Физический файл должен быть связан с логическим до выполнения процедур открытия файлов.

Текстовые файлы

Особое место в языке Pascal занимают текстовые файлы, компоненты которых имеют символьный тип. Для описания текстовых файлов в языке определен стандартный тип Text:

var TF1, TF2: Text;

Текстовые файлы представляют собой последовательность строк, а строки — последовательность символов. Строки имеют переменную длину, каждая строка завершается признаком конца строки.

Компонентные файлы

Компонентный, или типизированный файл, — это файл с объявленным типом его компонент. Компонентные файлы состоят из машинных представлений значений переменных, они хранят данные в том же виде, что и память ЭВМ.

Описание величин файлового типа имеет вид:

```
type M = File Of T;
```

где M — имя файлового типа;
T — тип компоненты.

Компонентами файла могут быть все скалярные типы, а из структурированных — массивы, множества, записи. Практически во всех конкретных реализациях языка Pascal конструкция «файл файлов» недопустима.

Все операции над компонентными файлами производятся с помощью стандартных процедур.

```
Write(f,X1,X2,...XK)
```

Бестиповые файлы

Бестиповые файлы позволяют записывать на диск произвольные участки памяти ЭВМ и считывать их с диска в память. Описываются бестиповые файлы следующим образом:

```
var f: File;
```

Теперь перечислим процедуры и функции для работы с различными видами файлов.

1. Procedure Assign(var F; FileName: String);

Процедура AssignFile сопоставляет имя внешнего файла с файловой переменной.

F — файловая переменная любого файлового типа, FileName — выражение типа String или выражение типа PChar, если допускается расширенный синтаксис. Все дальнейшие операции с F производятся с внешним файлом.

Нельзя использовать процедуру с уже открытой файловой переменной.

2. Procedure Close(var F);

Процедура разрывает связь между файловой переменной и внешним дисковым файлом и закрывает файл.

F — файловая переменная любого файлового типа, открытая процедурами Reset, Rewrite или Append. Внешний файл, связанный с F, полностью модифицируется и затем закрывается, освобождая дескриптор файла для повторного использования.

Директива {$I+} позволяет обрабатывать ошибки во время выполнения программы, используя обработку исключительных ситуаций. При выключенной директиве {$I–} необходимо использовать IOResult для проверки ошибок ввода-вывода.

3. Function Eof(var F): Boolean;

{Типизированные или нетипизированные файлы}

Function Eof[(var F: Text)]: Boolean;

{Текстовые файлы}

Проверяет, является или нет текущая позиция файла концом файла.

Eof(F) возвращает True, если текущая позиция файла находится за последним символом файла или если файл пуст; иначе Eof (F) возвращает False.

Директива {$I+} позволяет обрабатывать ошибки во время выполнения программы, используя обработку исключительных ситуаций. При выключенной директиве {$I–}, необходимо использовать IOResult для проверки ошибок ввода-вывода.

4. Procedure Erase(var F);

Удаляет внешний файл, связанный с F.

F — файловая переменная любого файлового типа.

Перед вызовом процедуры Erase файл необходимо закрыть.

Директива {$I+} позволяет обрабатывать ошибки во время выполнения программы, используя обработку исключительных ситуаций. При выключенной директиве {$I–}, необходимо использовать IOResult для проверки ошибок ввода-вывода.

5. Function FileSize(var F): Integer;

Возвращает размер в байтах файла F. Однако, если F — типизированный файл, FileSize возвратит число записей в файле. Перед

использованием функции FileSize файл должен быть открыт. Если файл пуст, FileSize(F) возвращает нуль.

F — переменная любого файлового типа.

6. Function FilePos(var F): LongInt;

Возвращает текущую позицию файла внутри файла.

Перед использованием функции FilePos, файл должен быть открыт. Функция FilePos не используется с текстовыми файлами. F — переменная любого файлового типа, кроме типа Text.

7. Procedure Reset(var F [: File; RecSize: Word]);

Открывает существующий файл.

F — переменная любого файлового типа, связанного с внешним файлом с помощью AssignFile. RecSize — необязательное выражение, которое используется, если F — нетипизированный файл. Если F — нетипизированный файл, RecSize определяет размер записи, который используется при передаче данных. Если RecSize опущен, заданный по умолчанию размер записи равен 128 байтов.

Процедура Reset открывает существующий внешний файл, ассоциированный с файловой переменной F. Если внешнего файла с таким именем нет, возникает ошибка времени выполнения. Если файл, связанный с F, уже открыт, он сначала закрывается и затем вновь открывается. Текущая позиция файла устанавливается к началу файла.

8. Procedure Rewrite(var F: File [; Recsize: Word]);

Создает и открывает новый файл.

F — переменная любого файлового типа, связанного с внешним файлом с использованием AssignFile. RecSize — необязательное выражение, которое используется, если F — нетипизированный файл. Если F — нетипизированный файл, RecSize определяет размер записи, который используется при передаче данных. Если RecSize опущен, заданный по умолчанию размер записи равен 128 байтов.

Процедура Rewrite создает новый внешний файл с именем, связанным с F. Если внешний файл с тем же самым именем уже существует, он удаляется, и создается новый пустой файл.

9. Procedure Seek(var F; N: LongInt);

Перемещает текущую позицию файла к определенному компоненту. Можно использовать процедуру только с открытыми типизированными или нетипизированными файлами.

Текущая позиция файла F перемещается к номеру N. Номер первого компонента файла — 0.

Инструкция Seek(F, FileSize(F)) перемещает текущую позицию файла в конец файла.

10. Procedure Append(var F: Text);

Открывает существующий текстовый файл для добавления информации в конец файла (дозапись).

Если внешнего файла с данным именем не существует, происходит ошибка времени выполнения. Если файл F уже открыт, он закрывается и вновь открывается. Текущая позиция файла устанавливается к концу файла.

11. Function Eoln[(var F: Text)]: Boolean;

Проверяет, является ли текущая позиция файла концом строки текстового файла.

Eoln(F) возвращает True, если текущая позиция файла — в конце строки или файла; иначе Eoln(F) возвращает False.

12. Procedure Read(F, V1 [, V2, ..., Vn]);

{Типизированные и нетипизированные файлы}

Procedure Read([var F: Text;] V1 [, V2, ..., Vn]);

{Текстовые файлы}

Для типизированных файлов процедура читает компонент файла в переменную. При каждом считывании текущая позиция в файле продвигается к следующему элементу.

Для текстовых файлов читается одно или несколько значений в одну или несколько переменных.

С переменными типа String Read считывает все символы вплоть до следующей метки конца строки (но не включая ее) или до тех пор пока функция Eof(F) не примет значение True. Пере-менной присваивается получившаяся в результате символьная строка.

В случае переменной целого или вещественного типа процедура ожидает поступления последовательности символов, образующих число по правилам синтаксиса языка Pascal. Считывание прекращается при обнаружении первого пробела, символа табуляции или метки конца строки либо в том случае, если функция Eof(F) принимает значение True. Если числовая строка не соответствует ожидаемому формату, то происходит ошибка ввода-вывода.

13. Procedure Readln([var F: Text;] V1 [, V2, ..., Vn]);

Является расширением процедуры Read и определена для текстовых файлов. Считывает строку символов в файле, включая маркер конца строки, и переходит к началу следующей строки. Вызов функции Readln(F) без параметров приводит к перемещению те-

кущей позиции файла на начало следующей строки, если она имеется, в противном случае происходит переход к концу файла.

14. Function SeekEof[(var F: Text)]: Boolean;

Возвращает признак конца файла и может использоваться только для открытых текстовых файлов. Обычно применяется для считывания числовых значений из текстовых файлов.

15. Function SeekEoln[(var F: Text)]: Boolean;

Возвращает признак конца строки в файле и может использоваться только для открытых текстовых файлов. Обычно применяется для считывания числовых значений из текстовых файлов.

16. Procedure Write([var F: Text;] P1 [, P2, ..., Pn]);

{Текстовые файлы}

Записывает одну или более величин в текстовый файл.

Каждый параметр записи должен иметь тип Char, один из целочисленных типов (Byte, ShortInt, Word, LongInt, Cardinal), один из типов с плавающей запятой (Single, Real, Double, Extended, Currency), один из строковых типов (PChar, AnsiString, ShortString), или один из логических типов (Boolean, Bool).

Procedure Write(F, V1, ..., Vn);

{Типизированные файлы}

Записывает переменную в компонент файла. Переменные V1,..., Vn должны быть того же типа, что и элементы файла. При каждой записи переменной текущая позиция в файле передвигается к следующему элементу.

17. Procedure Writeln([var F: Text;] [P1, P2, ..., Pn]);

{Текстовые файлы}

Выполняет операцию Write, затем помещает метку конца строки в файл.

Вызов Writeln(F) без параметров записывает в файл маркер конца строки. Файл должен быть открыт для вывода.

2. Модули. Виды модулей

Модуль(UNIT) в Pascal — это особым образом оформленная библиотека подпрограмм. Модуль, в отличие от программы, не может быть запущен на выполнение самостоятельно, он может только участвовать в построении программ и других модулей. Модули позволя-

ют создавать личные библиотеки процедур и функций и строить программы практически любого размера.

Модуль в Pascal представляет собой отдельно хранимую и независимо компилируемую программную единицу. В общем случае модуль — это совокупность программных ресурсов, предназначенных для использования другими программами. Под программными ресурсами понимаются любые элементы языка Pascal: константы, типы, переменные, подпрограммы. Модуль сам по себе не является выполняемой программой, его элементы используются другими программными единицами.

Все программные элементы модуля можно разбить на две части:

1) программные элементы, предназначенные для использования другими программами или модулями, такие элементы называют видимыми вне модуля;

2) программные элементы, необходимые только для работы самого модуля, их называют невидимыми (или скрытыми).

В соответствии с этим модуль, кроме заголовка, содержит три основные части, называемыми интерфейсной, исполнимой и инициализируемой.

В общем случае модуль имеет следующую структуру:

```
unit <имя модуля>; {заголовок модуля}
interface
{описание видимых программных элементов модуля}
implementation
{описание скрытых программных элементов модуля}
begin
{операторы инициализации элементов модуля}
end.
```

В частном случае модуль может не содержать части реализации и части инициализации, тогда структура модуля будет такой:

```
unit <имя модуля>; {заголовок модуля}
interface
{описание видимых программных элементов модуля}
implementation
end.
```

Использование в модулях процедур и функций имеет свои особенности. Заголовок подпрограммы содержит все сведения, необходимые для ее вызова: имя, перечень и тип параметров, тип результата для функций. Эта информация должна быть доступна для других программ и модулей. С другой стороны, текст подпрограммы, реализующий ее алгоритм, другими программами и модулями не может быть использован. Поэтому заголовки процедур и функций помещают в интерфейсную часть модуля, а текст — в часть реализации.

Интерфейсная часть модуля содержит только видимые (доступные для других программ и модулей) заголовки процедур и функций (без служебного слова forward). Полный текст процедуры или функции помещают в часть реализации, причем заголовок может не содержать списка формальных параметров.

Исходный текст модуля должен быть откомпилирован с помощью директивы Make подменю Compile и записан на диск. Результатом компиляции модуля является файл с расширением .TPU (Turbo Pascal Unit). Основное имя модуля берется из заголовка модуля.

Для подключения модуля к программе необходимо указать его имя в разделе описания модулей, например:

```
uses Crt, Graph;
```

В том случае, если имена переменных в интерфейсной части модуля и в программе, использующей этот модуль, совпадают, обращение будет происходить к переменной, описанной в программе. Для обращения к переменной, описанной в модуле, необходимо применить составное имя, состоящее из имени модуля и имени переменной, разделенных точкой. Использование составных имен применяется не только к именам переменных, а ко всем именам, описанным в интерфейсной части модуля.

Рекурсивное использование модулей запрещено.

Если в модуле имеется раздел инициализации, то операторы из этого раздела будут выполнены перед началом выполнения программы, в которой используется этот модуль.

Перечислим, какие бывают виды модулей.

1. Модуль SYSTEM.

Модуль SYSTEM реализует поддерживающие подпрограммы нижнего уровня для всех встроенных средств, таких как ввод-вы-

вод, работа со строками, операции с плавающей точкой и динамическое распределение памяти.

Модуль SYSTEM содержит все стандартные и встроенные процедуры и функции Pascal. Любая подпрограмма Pascal, не являющаяся частью стандартного Pascal и не находящаяся ни в каком другом модуле, содержится в модуле System. Этот модуль автоматически используется во всех программах, и его не требуется указывать в операторе uses.

2. Модуль DOS.

Модуль Dos реализует многочисленные процедуры и функции Pascal, которые эквивалентны наиболее часто используемым вызовам DOS, как, например, GetTime, SetTime, DiskSize и так далее.

3. Модуль CRT.

Модуль CRT реализует ряд мощных программ, предоставляющих полную возможность управления средствами компьютера PC, такими, как управление режимом экрана, расширенные коды клавиатуры, цвета, окна и звуковые сигналы. Модуль CRT может использоваться только в программах, работающих на персональных компьютерах IBM PC, PC AT, PS/2 фирмы IBM и полностью совместимых с ними.

Одним из основных преимуществ использования модуля CRT является большая скорость и гибкость при выполнении операций работы с экраном. Программы, не работающие с модулем CRT, выводят на экран информацию с помощью средств операционной системы DOS, что связано с дополнительными непроизводительными затратами. При использовании модуля CRT выводимая информация посылается непосредственно в базовую систему ввода-вывода (BIOS) или для еще более быстрых операций непосредственно в видеопамять.

4. Модуль GRAPH.

С помощью процедур и функций, входящих в этот модуль, можно создавать различные графические изображения на экране.

5. Модуль OVERLAY.

Модуль OVERLAY позволяет уменьшить требования к памяти программы DOS реального режима. Фактически можно писать программы, превышающие общий объем доступной памяти, поскольку в каждый момент в памяти будет находиться только часть программы.

ЛЕКЦИЯ № 7. Динамическая память

1. Ссылочный тип данных. Динамическая память. Динамические переменные

Статической переменной (статически размещенной) называется описанная явным образом в программе переменная, обращение к ней осуществляется по имени. Место в памяти для размещения статических переменных определяется при компиляции программы. В отличие от таких статических переменных в программах, написанных на языке Pascal, могут быть созданы динамические переменные. Основное свойство динамических переменных заключается в том, что они создаются, и память для них выделяется во время выполнения программы.

Размещаются динамические переменные в динамической области памяти (heap-области). Динамическая переменная не указывается явно в описаниях переменных, и к ней нельзя обратиться по имени. Доступ к таким переменным осуществляется с помощью указателей и ссылок.

Ссылочный тип (указатель) определяет множество значений, которые указывают на динамические переменные определенного типа, называемого базовым типом. Переменная ссылочного типа содержит адрес динамической переменной в памяти. Если базовый тип является еще не описанным идентификатором, то он должен быть описан в той же самой части описания типов, что и тип-указатель.

Зарезервированное слово nil обозначает константу со значением указателя, которая ни на что не указывает.

Приведем пример описания динамических переменных.

```
var p1, p2 : ^real;
  p3, p4 : ^integer;
  ...
```

2. Работа с динамической памятью. Нетипизированные указатели

Процедуры и функции работы с динамической памятью

1. Процедура New(var p:Pointer).

Выделяет место в динамической области памяти для размещения динамической переменной p^, и ее адрес присваивает указателю p.

2. Процедура Dispose(var p:Pointer).

Освобождает участок памяти, выделенный для размещения динамической переменной процедурой New, и значение указателя p становится неопределенным.

3. Процедура GetMem(var p:Pointer; size:Word).

Выделяет участок памяти в heap-области, присваивает адрес его начала указателю p, размер участка в байтах задается параметром size.

4. Процедура FreeMem(var p:Pointer; size:Word).

Освобождает участок памяти, адрес начала которого определен указателем p, а размер — параметром size. Значение указателя p становится неопределенным.

5. Процедура Mark(var p:Pointer) записывает в указатель p адрес начала участка свободной динамической памяти на момент ее вызова.

6. Процедура Release(var p:Pointer) освобождает участок динамической памяти, начиная с адреса, записанного в указатель p процедурой Mark, т. е. очищает ту динамическую память, которая была занята после вызова процедуры Mark.

7. Функция MaxAvail:Longint возвращает длину в байтах самого длинного свободного участка динамической памяти.

8. Функция MemAvail:Longint возвращает полный объем свободной динамической памяти в байтах.

9. Вспомогательная функция SizeOf(X):Word возвращает объем в байтах, занимаемый X, причем X может быть либо именем переменной любого типа, либо именем типа.

Встроенный тип Pointer обозначает нетипизированный указатель, т. е. указатель, который не указывает ни на какой определенный тип. Переменные типа Pointer могут быть разыменованы: указание символа ^ после такой переменной вызывает появление ошибки.

Как и значение, обозначаемое словом nil, значения типа Pointer совместимы со всеми другими типами указателей.

ЛЕКЦИЯ № 8.
Абстрактные структуры данных

1. Абстрактные структуры данных

Структурированные типы данных, такие как массивы, множества, записи, представляют собой статические структуры, так как их размеры неизменны в течение всего времени выполнения программы.

Часто требуется, чтобы структуры данных меняли свои размеры в ходе решения задачи. Такие структуры данных называются динамическими. К ним относятся стеки, очереди, списки, деревья и др.

Описание динамических структур с помощью массивов, записей и файлов приводит к неэкономному использованию памяти ЭВМ и увеличивает время решения задач.

Каждая компонента любой динамической структуры представляет собой запись, содержащую, по крайней мере, два поля: одно поле типа «указатель», а второе — для размещения данных. В общем случае запись может содержать не один, а несколько указателей и несколько полей данных. Поле данных может быть переменной, массивом, множеством или записью.

Если в указующей части содержится адрес одного элемента списка, то список называется однонаправленным (или односвязным). Если же он содержит две компоненты, то двусвязным. Над списками можно проводить различные операции, например:

1) добавление элемента к списку;
2) удаление элемента из списка с заданным ключом;
3) поиск элемента с заданным значением ключевого поля;
4) сортировка элементов списка;
5) деление списка на два и более списков;
6) объединение двух и более списков в один;
7) другие операции.

Однако, как правило, необходимости во всех операциях при решении различных задач не возникает. Поэтому в зависимости от основных операций, которые необходимо применить, существуют различные виды списков. Наиболее популярные из них — это стек и очередь.

2. Стеки

Стеком называется динамическая структура данных, добавление компоненты в которую и исключение компоненты из которой производится из одного конца, называемого вершиной стека. Стек работает по принципу LIFO(Last-In, First-Out) — «Поступивший последним, обслуживается первым».

Обычно над стеками выполняется три операции:

1) начальное формирование стека (запись первой компоненты);
2) добавление компоненты в стек;
3) выборка компоненты (удаление).

Для формирования стека и работы с ним необходимо иметь две переменные типа «указатель», первая из которых определяет вершину стека, а вторая — вспомогательная.

Пример. Составить программу, которая формирует стек, добавляет в него произвольное количество компонент, а затем читает все компоненты и выводит их на экран дисплея. В качестве данных взять строку символов. Ввод данных — с клавиатуры, признак конца ввода — строка символов END.

```
Program STACK;
uses Crt;
type
 Alfa = String[10];
 PComp = ^Comp;
 Comp = Record
 sD : Alfa;
 pNext : PComp
end;
var
 pTop : PComp;
 sC : Alfa;

Procedure CreateStack(var pTop : PComp; var sC : Alfa);
begin
 New(pTop);
 pTop^.pNext := NIL;
 pTop^.sD := sC;
end;
```

```
Procedure AddComp(var pTop : PComp; var sC : Alfa);
var pAux : PComp;
begin
 NEW(pAux);
 pAux^.pNext := pTop;
 pTop := pAux;
 pTop^.sD := sC;
end;

Procedure DelComp(var pTop : PComp; var sC : ALFA);
begin
 sC := pTop^.sD;
 pTop := pTop^.pNext;
end;

begin
 Clrscr;
 writeln(' ВВЕДИ СТРОКУ ');
 readln(sC);
 CreateStack(pTop, sC);
 repeat
 writeln(' ВВЕДИ СТРОКУ ');
 readln(sC);
 AddComp(pTop, sC);
 until sC = 'END';

 writeln('****** ВЫВОД РЕЗУЛЬТАТОВ ******');
 repeat
 DelComp(pTop, sC);
 writeln(sC);
 until pTop = NIL;
end.
```

3. Очереди

Очередью называется динамическая структура данных, добавление компоненты в которую производится в один конец, а выборка осуществляется с другого конца. Очередь работает по прин-

ципу FIFO (First-In, First-Out) — «Поступивший первым, обслуживается первым».

Для формирования очереди и работы с ней необходимо иметь три переменные типа указатель, первая из которых определяет начало очереди, вторая — конец очереди, третья — вспомогательная.

Пример. Составить программу, которая формирует очередь, добавляет в нее произвольное количество компонент, а затем читает все компоненты и выводит их на экран дисплея. В качестве данных взять строку символов. Ввод данных — с клавиатуры, признак конца ввода — строка символов END.

```
Program QUEUE;
uses Crt;
type
 Alfa = String[10];
 PComp = ^Comp;
 Comp = record
 sD : Alfa;
 pNext : PComp;
end;
var
 pBegin, pEnd : PComp;
 sC : Alfa;

Procedure CreateQueue(var pBegin,pEnd:PComp; var sC:Alfa);
begin
 New(pBegin);
 pBegin^.pNext := NIL;
 pBegin^.sD := sC;
 pEnd := pBegin;
end;

Procedure AddQueue(var pEnd : PComp; var sC : Alfa);
var pAux : PComp;
begin
 New(pAux);
 pAux^.pNext := NIL;
 pEnd^.pNext := pAux;
 pEnd := pAux;
```

```
 pEnd^.sD := sC;
end;

Procedure DelQueue(var pBegin : PComp; var sC : Alfa);
begin
 sC := pBegin^.sD;
 pBegin := pBegin^.pNext;
end;

begin
 Clrscr;
 writeln(' ВВЕДИ СТРОКУ ');
 readln(sC);
 CreateQueue(pBegin, pEnd, sC);

 repeat
 writeln(' ВВЕДИ СТРОКУ ');
 readln(sC);
 AddQueue(pEnd, sC);
 until sC = 'END';

 writeln(' ***** ВЫВОД РЕЗУЛЬТАТОВ *****');
 repeat
 DelQueue(pBegin, sC);
 writeln(sC);
 until pBegin = NIL;
end.
```

ЛЕКЦИЯ № 9. Древовидные структуры данных

1. Древовидные структуры данных

Древовидной структурой данных называется конечное множество элементов-узлов, между которыми существуют отношения — связь исходного и порожденного.

Если использовать рекурсивное определение, предложенное Н. Виртом, то древовидная структура данных с базовым типом t — это либо пустая структура, либо узел типа t, с которым связано конечное множество древовидных структур с базовым типом t, называемых поддеревьями.

Далее дадим определения, используемые при оперировании древовидными структурами.

Если узел y находится непосредственно под узлом x, то узел y называется непосредственным потомком узла x, а x — непосредственным предком узла y, т. е., если узел x находится на i-ом уровне, то соответственно узел y находится на (i + 1)-ом уровне.

Максимальный уровень узла дерева называется высотой или глубиной дерева. Предка не имеет только один узел дерева — его корень.

Узлы дерева, у которых не имеется потомков, называются терминальными узлами (или листами дерева). Все остальные узлы называются внутренними узлами. Количество непосредственных потомков узла определяет степень этого узла, а максимально возможная степень узла в данном дереве определяет степень дерева.

Предков и потомков нельзя поменять местами, т. е. связь исходного и порожденного действует только в одном направлении.

Если пройти от корня дерева к некоторому конкретному узлу, то количество ветвей дерева, которое при этом будет пройдено, называется длиной пути для этого узла. Если все ветви (узлы) у дерева упорядочены, то дерево называется упорядоченным.

Частным случаем древовидных структур являются бинарные деревья. Это деревья, в которых каждый потомок имеет не более двух потомков, называемых левым и правым поддеревьями. Таким образом, бинарное дерево — это древовидная структура, степень которой равна двум.

Упорядоченность бинарного дерева определяется по следующему правилу: каждому узлу соответствует свое ключевое поле, и для каждого узла значение ключа больше всех ключей в его левом поддереве и меньше всех ключей в его правом поддереве.

Дерево, степень которого больше двух, называется сильноветвящимся.

2. Операции над деревьями

Далее будем рассматривать все операции применительно к бинарным деревьям.

I. Построение дерева.

Приведем алгоритм построения упорядоченного дерева.

1. Если дерево пусто, то данные переносятся в корень дерева. Если же дерево не пусто, то осуществляется спуск по одной из его ветвей таким образом, чтобы упорядоченность дерева не нарушалась. В результате новый узел становится очередным листом дерева.

2. Чтобы добавить узел в уже существующее дерево, можно воспользоваться вышеприведенным алгоритмом.

3. При удалении узла из дерева следует быть внимательным. Если удаляемый узел является листом, или же имеет только одного потомка, то операция проста. Если же удаляемый узел имеет двух потомков, то необходимо будет найти узел среди его потомков, который можно будет поставить на его место. Это нужно в силу требования упорядоченности дерева.

Можно поступить таким образом: поменять удаляемый узел местами с узлом, имеющем самое большое значение ключа в левом поддереве, или с узлом, имеющем самое малое значение ключа в правом поддереве, а затем удалить искомый узел как лист.

II. Поиск узла с заданным значением ключевого поля.

При осуществлении этой операции необходимо совершить обход дерева. Необходимо учитывать различные формы записи дерева: префиксную, инфиксную и постфиксную.

Возникает вопрос: каким образом представить узлы дерева, чтобы было наиболее удобно работать с ними? Можно представлять дерево с помощью массива, где каждый узел описывается величиной комбинированного типа, у которой информационное поле символьного типа и два поля ссылочного типа. Но это не совсем удобно, так как деревья имеют большое количество узлов, заранее не определенное. Поэтому лучше всего при описании дерева ис-

пользовать динамические переменные. Тогда каждый узел представляется величиной одного типа, которая содержит описание заданного количества информационных полей, а количество соответствующих полей должно быть равно степени дерева. Логично отсутствие потомков определять ссылкой nil. Тогда на языке Pascal описание бинарного дерева может выглядеть следующим образом:

```
TYPE TreeLink = ^Tree;
Tree = record;
Inf : <тип данных>;
Left, Right : TreeLink;
End.
```

3. Примеры реализации операций

1. Построить дерево из n узлов минимальной высоты, или идеально сбалансированное дерево (количество узлов левого и правого поддеревьев такого дерева должны отличаться не более чем на единицу).

Рекурсивный алгоритм построения:

1) первый узел берется в качестве корня дерева.
2) тем же способом строится левое поддерево из nl узлов.
3) тем же способом строится правое поддерево из nr узлов;

$$nr = n - nl - 1.$$

В качестве информационного поля будем брать номера узлов, вводимые с клавиатуры. Рекурсивная функция, реализующая данное построение, будет выглядеть следующим образом:

```
Function Tree(n : Byte) : TreeLink;
Var t : TreeLink; nl,nr,x : Byte;
Begin
 If n = 0 then Tree := nil
 Else
 Begin
 nl := n div 2;
 nr = n – nl – 1;
 writeln('Введите номер вершины ');
 readln(x);
 new(t);
```

```
t^.inf := x;
t^.left := Tree(nl);
t^.right := Tree(nr);
Tree := t;
End;
{Tree}
End.
```

2. В бинарном упорядоченном дереве найти узел с заданным значением ключевого поля. Если такого элемента в дереве нет, то добавить его в дерево.

```
Procedure Search(x : Byte; var t : TreeLink);
Begin
If t = nil then
Begin
New(t);
t^inf := x;
t^.left := nil;
t^.right := nil;
End
Else if x < t^.inf then
Search(x, t^.left)
Else if x > t^.inf then
Search(x, t^.right)
Else
Begin
{обработка найденного элемента}
...
End;
End.
```

3. Написать процедуры обхода дерева в прямом, симметричном и обратном порядке соответственно.

```
3.1. Procedure Preorder(t : TreeLink);
Begin
If t <> nil then
Begin
Writeln(t^.inf);
```

```
Preorder(t^.left);
Preorder(t^.right);
End;
End;
```

```
3.2. Procedure Inorder(t : TreeLink);
Begin
If t <> nil then
Begin
Inorder(t^.left);
Writeln(t^.inf);
Inorder(t^.right);
End;
End.
```

```
3.3. Procedure Postorder(t : TreeLink);
Begin
If t <> nil then
Begin
Postorder(t^.left);
Postorder(t^.right);
Writeln(t^.inf);
End;
End.
```

4. В бинарном упорядоченном дереве удалить узел с заданным значением ключевого поля.

Опишем рекурсивную процедуру, которая будет учитывать наличие требуемого элемента в дереве и количество потомков этого узла. Если удаляемый узел имеет двух потомков, то он будет заменен самым большим значением ключа в его левом поддереве, и только после этого он будет окончательно удален.

```
Procedure Delete1(x : Byte; var t : TreeLink);
Var p : TreeLink;
Procedure Delete2(var q : TreeLink);
Begin
If q^.right <> nil then Delete2(q^.right)
Else
Begin
```

```
p^.inf := q^.inf;
p := q;
q := q^.left;
End;
End;
Begin
If t = nil then
Writeln('искомого элемента нет')
Else if x < t^.inf then
Delete1(x, t^.left)
Else if x > t^.inf then
Delete1(x, t^.right)
Else
Begin
P := t;
If p^.left = nil then
t := p^.right
Else
If p^.right = nil then
t := p^.left
Else
Delete2(p^.left);
End;
End.
```

ЛЕКЦИЯ № 10. Графы

1. Понятие графа. Способы представления графа

Граф — пара G = (V,E), где V — множество объектов произвольной природы, называемых вершинами, а E — семейство пар ei = (vi1, vi2), vijOV, называемых *ребрами*. В общем случае множество V и (или) семейство E могут содержать бесконечное число элементов, но мы будем рассматривать только конечные графы, т.е. графы, у которых как V, так и E конечны. Если порядок элементов, входящих в ei, имеет значение, то граф называется *ориентированным*, сокращенно — орграф, иначе — *неориентированным*. Ребра орграфа называются *дугами*. В дальнейшем будем считать, что термин «граф», применяемый без уточнений (ориентированный или неориентированный), обозначает неориентированный граф.

Если e = <u,v>, то вершины v и u называются концами ребра. При этом говорят, что ребро e является смежным (инцидентным) каждой из вершин v и u. Вершины v и u также называются *смежными (инцидентными)*. В общем случае допускаются ребра вида e = <v, v>; такие ребра называются *петлями*.

Степень вершины графа — это число ребер, инцидентных данной вершине, причем петли учитываются дважды. Поскольку каждое ребро инцидентно двум вершинам, сумма степеней всех вершин графа равна удвоенному количеству ребер: Sum(deg(vi), i=1...|V|) = 2 * |E|.

Вес вершины — число (действительное, целое или рациональное), поставленное в соответствие данной вершине (интерпретируется как стоимость, пропускная способность и т. д.). Вес, длина ребра — число или несколько чисел, которые интерпретируются как длина, пропускная способность и т. д.

Путем в графе (или маршрутом в орграфе) называется чередующаяся последовательность вершин и ребер (или дуг — в орграфе) вида v0, (v0,v1), v1, ... , (vn – 1,vn), vn. Число n называется длиной пути. Путь без повторяющихся ребер называется цепью, без повторяющихся вершин — простой цепью. Путь может быть

замкнутым ($v0 = vn$). Замкнутый путь без повторяющихся ребер называется *циклом* (или контуром в орграфе); без повторяющихся вершин (кроме первой и последней) — простым циклом.

Граф называется связным, если существует путь между любыми двумя его вершинами, и несвязным — в противном случае. Несвязный граф состоит из нескольких связных компонент (связных подграфов).

Существуют различные способы представления графов. Рассмотрим каждый из них в отдельности.

1. Матрица инцидентности.

Это прямоугольная матрица размерности $n \times m$, где n — количество вершин, а m — количество ребер. Значения элементов матрицы определяются следующим образом: если ребро xi и вершина vj инцидентны, то значение соотвествующего элемента матрицы равно единице, в противном случае значение равно нулю. Для ориентированных графов матрица инцидентности строится по следующему принципу: значение элемента равно –1, если ребро xi исходит из вершины vj, равно 1, если ребро xi заходит в вершину vj, и равно 0 в противном случае.

2. Матрица смежности.

Это квадратная матрица размерности $n \times n$, где n — количество вершин. Если вершины vi и vj смежны, т. е. если существует ребро, их соединяющее, то соответствующий элемент матрицы равен единице, в противном случае он равен нулю. Правила построения данной матрицы для ориентированного и неориентированного графов не отличаются. Матрица смежности более компактна, чем матрица инцидентности. Следует заметить, что эта матрица также сильно разрежена, однако в случае неориентированного графа она является симметричной относительно главной диагонали, поэтому можно хранить не всю матрицу, а только ее половину (треугольную матрицу).

3. Список смежности (инцидентности).

Представляет собой структуру данных, которая для каждой вершины графа хранит список смежных с ней вершин. Список представляет собой массив указателей, i-ый элемент которого содержит указатель на список вершин, смежных с i-ой вершиной.

Список смежности более эффективен по сравнению с матрицей смежности, так как исключает хранение нулевых элементов.

4. Список списков.

Представляет собой древовидную структуру данных, в которой одна ветвь содержит списки вершин, смежных для каждой из

вершин графа, а вторая ветвь указывает на очередную вершину графа. Такой способ представления графа является наиболее оптимальным.

2. Представление графа списком инцидентности. Алгоритм обхода графа в глубину

Для реализации графа в виде списка инцидентности можно использовать следующий тип:

```
Type List = ^S;
  S = record;
  inf : Byte;
  next : List;
  end;
```

Тогда граф задается следующим образом:

```
Var Gr : array[1..n] of List;
```

Теперь обратимся к процедуре обхода графа. Это вспомогательный алгоритм, который позволяет просмотреть все вершины графа, проанализировать все информационные поля. Если рассматривать обход графа в глубину, то существуют два типа алгоритмов: рекурсивный и нерекурсивный.

При рекурсивном алгоритме обхода графа в глубину мы берем произвольную вершину u, отыскиваем произвольную непросмотренную (новую) вершину v, смежную с ней. Затем принимаем вершину v за неновую и отыскиваем любую смежную с ней новую вершину. Если же у какой-либо вершины нет более новых непросмотренных вершин, то полагаем эту вершину использованной и возвращаемся на уровень выше в ту вершину, из которой попали в нашу использованную вершину. Обход продолжается таким образом до тех пор, пока в графе не останется новых непросмотренных вершин.

На языке Pascal процедура обхода в глубину будет выглядеть следующим образом:

```
Procedure Obhod(gr : Graph; k : Byte);
Var g : Graph; l : List;
```

```
Begin
nov[k] := false;
g := gr;
While g^.inf <> k do
g := g^.next;
l := g^.smeg;
While l <> nil do begin
If nov[l^.inf] then Obhod(gr, l^.inf);
l := l^.next;
End;
End;
```

Примечание

В данной процедуре при описании типа Graph имелось в виду описание графа списком списков. Массив nov[i] — специальный массив, i-ый элемент которого равен True, если i-ая вершина не просмотрена, и False — в противном случае.

Также часто используется нерекурсивный алгоритм обхода. В этом случае рекурсия заменяется на стек. Как только вершина просмотрена, она помещается в стек, а использованной она становится, когда больше нет новых вершин, смежных с ней.

3. Представление графа списком списков. Алгоритм обхода графа в ширину

Граф можно определить с помощью списка списков следующим образом:

```
Type List = ^Tlist;
Tlist = record
inf : Byte;
next : List;
 end;
Graph = ^TGpaph;
TGpaph = record
inf : Byte;
smeg : List;
next : Graph;
end;
```

При обходе графа в ширину мы выбираем произвольную вершину и просматриваем сразу все вершины, смежные с ней. Вместо стека используется очередь. Алгоритм обхода в ширину очень удобен при нахождении наикратчайшего пути в графе.

Приведем процедуру обхода графа в ширину на псевдокоде:

```
Procedure Obhod2(v);
{величины spisok, nov — глобальные}
Begin
queue = O;
queue <= v;
nov[v] = False;
While queue <> O do
Begin
p <= queue;
For u in spisok(p) do
If nov[u] then
Begin
nov[u] := False;
queue <= u;
End;
End;
End;
```

ЛЕКЦИЯ № 11. Объектный тип данных

1. Объектный тип в Pascal. Понятие объекта, его описание и использование

Исторически первым подходом в программировании являлось процедурное программирование, иначе называемое программированием снизу вверх. Вначале создавались общие библиотеки стандартных программ, используемые в различных областях применения ЭВМ. Затем на основе этих программ создавались более сложные программы для решения конкретных задач.

Однако вычислительная техника постоянно развивалась, ее стали применять для решения различных задач производства, экономики, в связи с чем возникла необходимость обработки данных различного формата и решения нестандартных задач (например, нечислового характера). Поэтому при разработке языков программирования стали обращать внимание на создание различных типов данных. Это способствовало появлению таких сложных типов данных, как комбинированные, множественные, строковые, файловые и др. Прежде чем решать задачу, программист проводил декомпозицию, т. е. разбиение задачи на несколько подзадач, для каждой из которых писался свой отдельный модуль. Основная технология программирования включала в себя три этапа:

1) проектирование сверху вниз;
2) модульное программирование;
3) структурное кодирование.

Но начиная с середины 60-х г. XX в., стали формироваться новые понятия и подходы, которые легли в основу технологии объектно-ориентированного программирования. В данном подходе осуществляется моделирование и описание реального мира на уровне понятий конкретной предметной области, к которой относится решаемая задача.

Объектно-ориентированное программирование представляет собой метод программирования, который весьма близко напоминает наше поведение. Оно является естественной эволюцией более ранних нововведений в разработке языков программирова-

ния. Объектно-ориентированное программирование является более структурным, чем все предыдущие разработки, касающиеся структурного программирования. Оно также является более модульным и более абстрактным, чем предыдущие попытки абстрагирования данных и переноса деталей программирования на внутренний уровень. Объектно-ориентированный язык программирования характеризуется тремя основными свойствами:

1) Инкапсуляцией. Комбинирование записей с процедурами и функциями, манипулирующими полями этих записей, формирует новый тип данных — объект;
2) Наследованием.Определение объекта и его дальнейшее использование для построения иерархии порожденных объектов с возможностью для каждого порожденного объекта, относящегося к иерархии, доступа к коду и данным всех порождающих объектов;
3) Полиморфизмом. Присваивание действию одного имени, которое затем совместно используется вниз и вверх по иерархии объектов, причем каждый объект иерархии выполняет это действие способом, именно ему подходящим.

Говоря об объекте, мы вводим в рассмотрение новый тип данных — объектный. Объектный тип является структурой, состоящей из фиксированного числа компонентов. Каждый компонент является либо полем, содержащим данные строго определенного типа, либо методом, выполняющим операции над объектом. По аналогии с описанием переменных описание поля указывает тип данных этого поля и идентификатор, именующий поле: по аналогии с описанием процедуры или функции описание метода указывает заголовок процедуры, функции, конструктора или деструктора.

Объектный тип может наследовать компоненты другого объектного типа. Если тип T2 наследует от типа T1, то тип T2 является потомком типа T1, а сам тип T1 является родителем типа T2. Наследование является транзитивным, т. е. если T3 наследует от T2, а T2 наследует от T1, то T3 наследует от T1. Область (домен) объектного типа состоит из него самого и из всех его наследников.

Следующий исходный код приводит пример описания объектного типа.

```
type
 Point = object
 X, Y: integer;
 end;

 Rect = object
 A , B: TPoint;
 procedure Init(XA, YA, XB, YB: Integer);
 procedure Copy(var R: TRectangle);
 procedure Move(DX, DY: Integer);
 procedure Grow(DX, DY: Integer);
 procedure Intersect(var R: TRectangle);
 procedure Union(var R: TRectangle);
 function Contains(P: Point): Boolean;
 end;

 StringPtr = ^String;
 FieldPtr = ^TField;
 TField = object
 X, Y, Len: Integer;
 Name: StringPtr;
 constructor Copy(var F: TField);
 constructor Init(FX, FY, FLen: Integer; FName: String);
 destructor Done; virtual;
 procedure Display; virtual;
 procedure Edit; virtual;
 function GetStr: String; virtual;
 function PutStr(S: String): Boolean; virtual;
 end;

 StrFieldPtr = ^TStrField;
 StrField = object(TField)
 Value: PString;
 constructor Init(FX, FY, FLen: Integer; FName: String);
 destructor Done; virtual;
 function GetStr: String; virtual;
 function PutStr(S: String): Boolean;
 virtual;
```

```
function Get: string;
procedure Put(S: String);
end;

NumFieldPtr = ^TNumField;
TNumField = object(TField)
private
Value, Min, Max: Longint;
public
constructor Init(FX, FY, FLen: Integer; FName: String;
FMin, FMax: Longint);
function GetStr: String; virtual;
function PutStr(S: String): Boolean; virtual;
function Get: Longint;
function Put(N: Longint);
end;

ZipFieldPtr = ^TZipField;
ZipField = object(TNumField)
function GetStr: String; virtual;
function PutStr(S: String): Boolean;
virtual;
end.
```

В отличие от других типов объектные типы могут описываться только в разделе описаний типов, находящемся на самом внешнем уровне области действия программы или модуля. Таким образом, объектные типы не могут описываться в разделе описаний переменных или внутри блока процедуры, функции или метода.

Тип компоненты файлового типа не может иметь объектный тип или любой структурный тип, содержащий компоненты объектного типа.

2. Наследование

Процесс, с помощью которого один тип наследует характеристики другого типа, называется наследованием. Наследник называется порожденным (дочерним) типом, а тип, которому наследует дочерний тип, называется порождающим (родительским) типом.

Ранее известные типы записей Pascal не могут наследовать. Однако Borland Pascal расширяет язык Pascal для поддержки наследования. Одним из этих расширений является новая категория структуры данных, связанная с записями, но значительно более мощная. Типы данных в этой новой категории определяются с помощью нового зарезервированного слова «object». Тип объекта может быть определен как полный, самостоятельный тип в манере описания записей Pascal, но он может определяться и как потомок существующего типа объекта путем помещения порождающего (родительского) типа в скобки после зарезервированного слова «object».

3. Создание экземпляров объектов

Экземпляр объекта создается посредством описания переменной или константы объектного типа или путем применения стандартной процедуры New к переменной типа «указатель на объектный тип». Результирующий объект называется экземпляром объектного типа;

```
var
F: TField;
Z: TZipField;
FP: PField;
ZP: PZipField;
```

С учетом этих описаний переменных, F является экземпляром TField, а Z — экземпляром TZipField. Аналогично, после применения New к FP и ZP FP будет указывать на экземпляр TField, а ZP — на экземпляр TZipField.

Если объектный тип содержит виртуальные методы, то экземпляры этого объектного типа должны инициализироваться посредством вызова конструктора перед вызовом любого виртуального метода.

Ниже приведен пример:

```
var
 S: StrField;
 egin
 S.Init (1, 1, 25, 'Первое имя');
 S.Put ('Владимир');
 S.Display;
```

```
...
S.Done;
end.
```

Если S.Init не вызывался, то вызов S.Display приведет к неудачному завершению данного примера.

Присваивание экземпляра объектного типа не подразумевает инициализации экземпляра. Объект инициализируется кодом, генерируемым компилятором, который выполняется между вызовом конструктора и моментом когда выполнение фактически достигает первого оператора в блоке кода конструктора.

Если экземпляр объекта не инициализируется и проверка диапазона включена (директивой {$R+}), то первый вызов виртуального метода экземпляра объекта дает ошибку этапа выполнения. Если проверка диапазона выключена (директивой {$R–}), то первый вызов виртуального метода неинициализированного объекта может привести к непредсказуемому поведению.

Правило обязательной инициализации применимо также к экземплярам, которые являются компонентами структурных типов. Например:

```
var
 Comment: array [1..5] of TStrField;
 I: integer;
begin
 for I := 1 to 5 do
 Comment [I].Init (1, I + 10, 40, 'первое_имя');
  .
  .
  .
 for I := 1 to 5 do Comment [I].Done;
end;
```

Для динамических экземпляров инициализация, как правило, связана с размещением, а очистка — с удалением, что достигается благодаря расширенному синтаксису стандартных процедур New и Dispose. Например:

```
var
SP: StrFieldPtr;
```

```
begin
 New (SP, Init (1, 1, 25, 'первое_имя');
 SP^.Put ('Владимир');
 SP^.Display;
  .
  .
  .
 Dispose (SP, Done);
end.
```

Указатель на объектный тип является совместимым по присваиванию с указателем на любой родительский объектный тип, поэтому во время выполнения программы указатель на объектный тип может указывать на экземпляр этого типа или на экземпляр любого дочернего типа.

Например, указатель типа ZipFieldPtr может присваиваться указателям типа PZipField, PNumField и PField, а во время выполнения программы указатель типа PField может либо иметь значение nil, либо указывать на экземпляр TField, TNumField, или TZipField, или на любой экземпляр дочернего по отношению к TField типа.

Эти правила совместимости указателей по присваиванию применимы также к параметрам — переменным объектного типа. Например, методу TField.Copy могут быть переданы экземпляры типов TField, TStrField, TNumField, TZipField или любые другие экземпляры дочернего от TField типа.

4. Компоненты и область действия

Область действия идентификатора компоненты простирается за пределы объектного типа. Более того, область действия идентификатора компонента простирается сквозь блоки процедур, функций, конструкторов и деструкторов, которые реализуют методы объектного типа и его наследников. Исходя из этих соображений написание идентификатора компоненты должно быть уникальным внутри объектного типа и внутри всех его наследников, а также внутри всех его методов.

Область действия идентификатора компонента, описанного в части private описания типа, ограничивается модулем (программой), которая содержит описание объектного типа. Другими словами, частные (private) компоненты-идентификаторы действуют

как обычные общедоступные идентификаторы в рамках модуля, который содержит описание объектного типа, а вне модуля любые частные компоненты и идентификаторы неизвестны и недоступны. Поместив в один модуль связанные типы объектов, можно сделать так, что эти объекты смогут обращаться к частным компонентам друг друга, и эти частные компоненты будут неизвестны другим модулям.

В описании объектного типа заголовок метода может задавать параметры описываемого объектного типа, даже если описание еще неполное.

Лекция № 12. Методы

1. Методы

Описание метода внутри объектного типа соответствует опережающему описанию метода (forward). Таким образом, где-нибудь после описания объектного типа, но внутри той же самой области действия, что и область действия описания объектного типа, метод должен реализоваться путем определения его описания.

Для процедурных и функциональных методов определяющее описание имеет форму обычного описания процедуры или функции с тем исключением, что в этом случае идентификатор процедуры или функции рассматривается как идентификатор метода.

Для методов конструкторов и деструкторов определяющее описание принимает форму описания процедурного метода с тем исключением, что зарезервированное слово procedure заменяется зарезервированным словом constructor или destructor.

Определяющее описание метода может повторять (но не обязательно) список формальных параметров заголовка метода в объектном типе. В этом случае заголовок метода должен в точности повторять заголовок в объектном типе в порядке, типах и именах параметров и в типе возвращаемого функцией результата, если метод является функцией.

В определяющем описании метода всегда присутствует неявный параметр с идентификатором Self, соответствующий формальному параметру-переменной, обладающему объектным типом. Внутри блока метода Self представляет экземпляр, компонент метода которого был указан для активизации метода. Таким образом, любые изменения значений полей Self отражаются на экземпляре.

Область действия идентификатора компонента объектного типа распространяется на блоки процедур, функций, конструкторов и деструктора, которые реализуют методы данного объектного типа. Эффект получается тот же, как если бы в начало блока метода был вставлен *оператор with* в следующей форме:

```
with Self do
```

```
begin
 ...
end;
```

Исходя из этих соображений написание идентификаторов компонентов, формальных параметров метода, Self и любого идентификатора, введенного в исполняемую часть метода, должно быть уникальным.

Если требуется уникальный идентификатор метода, то используется уточненный идентификатор метода. Он состоит из идентификатора типа объекта, за которым следуют точка и идентификатор метода. Как и любому другому идентификатору, идентификатору уточненного метода, если требуется, могут предшествовать идентификатор пакета и точка.

Виртуальные методы

По умолчанию методы являются статическими, однако они могут, за исключением конструкторов, быть виртуальными (посредством включения *директивы virtual* в описание метода). Компилятор разрешает ссылки на вызовы статических методов во время процесса компиляции, тогда как вызовы виртуальных методов разрешаются во время выполнения. Это иногда называют поздним связыванием.

Если объектный тип объявляет или наследует какой-либо виртуальный метод, то переменные этого типа должны быть инициализированы посредством вызова конструктора перед вызовом любого виртуального метода. Таким образом, объектный тип, который описывает или наследует виртуальный метод, должен также описывать или наследовать, по крайней мере, один метод-конструктор.

Объектный тип может переопределять любой из методов, которые он наследует от своих родителей. Если описание метода в потомке указывает тот же идентификатор метода, что и описание метода в родителе, то описание в потомке переопределяет описание в родителе. Область действия переопределяющего метода расширяется до сферы действия потомка, в котором этот метод был введен, и будет оставаться таковой, пока идентификатор метода не будет переопределен снова.

Переопределение статического метода не зависит от изменения заголовка метода. В противоположность этому, переопределение виртуального метода должно сохранять порядок, типы и имена параметров, а также типы результатов функций, если таковые имеются. Более того, переопределение опять же должно включать *директиву virtual*.

Динамические методы

Borland Pascal поддерживает дополнительные методы с поздним связыванием, которые называются динамическими методами. Динамические методы отличаются от виртуальных только характером их диспетчеризации на этапе выполнения. Во всех других отношениях динамические методы считаются эквивалентными виртуальным.

Описание динамического метода эквивалентно описанию виртуального метода, но описание динамического метода должно включать в себя индекс динамического метода, который указывается непосредственно за ключевым словом virtual. Индекс динамического метода должен быть целочисленной константой в диапазоне от 1 до 656535 и должен быть уникальным среди индексов других динамических методов, содержащихся в объектном типе или его предках. Например:

```
procedure FileOpen(var Msg: TMessage); virtual 100;
```

Переопределение динамического метода должно соответствовать порядку, типам и именам параметров и точно соответствовать типу результата функции порождающего метода. Переопределение также должно включать в себя *директиву virtual*, за которой следует тот же индекс динамического метода, который был задан в объектном типе предка.

2. Конструкторы и деструкторы

Конструкторы и деструкторы являются специализированными формами методов. Используемые в связи с расширенным синтаксисом стандартных процедур New и Dispose конструкторы и деструкторы обладают способностью размещения и удаления динамических объектов. Кроме того, конструкторы имеют воз-

можность выполнить требуемую инициализацию объектов, содержащих виртуальные методы. Как и все другие методы, конструкторы и деструкторы могут наследоваться, а объекты могут содержать любое число конструкторов и деструкторов.

Конструкторы используются для инициализации вновь созданных объектов. Обычно инициализация основывается на значениях, передаваемых конструктору в качестве параметров. Конструктор не может быть виртуальным, так как механизм диспетчеризации виртуального метода зависит от конструктора, который первым совершил инициализацию объекта.

Приведем несколько примеров конструкторов:

```
constructor Field.Copy(var F: Field);
begin
Self := F;
end;

constructor Field.Init(FX, FY, FLen: integer; FName: string);
begin
X := FX;
Y := FY;
GetMem(Name, Length (FName) + 1);
Name^ := FName;
end;

constructor TStrField.Init(FX, FY, FLen: integer; FName: string);
begin
inherited Init(FX, FY, FLen, FName);
Field.Init(FX, FY, FLen, FName);
GetMem(Value, Len);
Value^ := '';
end;
```

Главным действием конструктора порожденного (дочернего) типа, такого как указанный выше TStrField.Init, почти всегда является вызов соответствующего конструктора его непосредственного родителя для инициализации наследуемых полей объекта. После выполнения этой процедуры конструктор инициализирует поля объекта, которые принадлежат только порожденному типу.

Деструкторы являются противоположностями конструкторов и используются для очистки объектов после их использования. Обычно очистка состоит в удалении всех полей указателей в объекте.

Примечание

Деструктор может быть виртуальным и часто является таковым. Деструктор редко имеет параметры.

Приведем несколько примеров деструкторов:

```
destructor Field.Done;
begin
 FreeMem(Name, Length (Name^) + 1);
end;

destructor StrField.Done;
begin
 FreeMem(Value, Len);
 Field.Done;
end;
```

Деструктор дочернего типа, такой как указанный выше TStrField. Done, обычно сначала удаляет введенные в порожденном типе поля указателей, а затем в качестве последнего действия вызывает соответствующий сборщик-деструктор непосредственного родителя для удаления унаследованных полей указателей объекта.

3. Деструкторы

Borland Pascal предоставляет специальный тип метода, называемый сборщиком мусора (или деструктором) для очистки и уда-ления динамически размещенного объекта. Деструктор объединяет шаг удаления объекта с какими-либо другими действиями или задачами, необходимыми для данного типа объекта. Для единственного типа объекта можно определить несколько деструкторов.

Деструктор определяется совместно со всеми другими методами объекта в определении типа объекта:

```
type
TEmployee = object
Name: string[25];
Title: string[25];
```

```
 Rate: Real;
constructor Init(AName, ATitle: String; ARate: Real);
destructor Done; virtual;
 function GetName: String;
 function GetTitle: String;
 function GetRate: Rate; virtual;
 function GetPayAmount: Real; virtual;
end;
```

Деструкторы можно наследовать, и они могут быть либо статическими, либо виртуальными. Поскольку различные программы завершения, как правило, требуют различные типы объектов, обычно рекомендуется, чтобы деструкторы всегда были виртуальными, благодаря чему для каждого типа объекта будет выполнен правильный деструктор.

Зарезервированное слово *destructor* не требуется указывать для каждого метода очистки, даже если определение типа объекта содержит виртуальные методы. Деструкторы в действительности работают только с динамически размещенными объектами.

При очистке динамически размещенного объекта деструктор осуществляет специальные функции: он гарантирует, что в динамически распределяемой области памяти всегда будет освобождаться правильное число байтов. Не может быть никаких опасений по поводу использования деструктора применительно к статически размещенным объектам; фактически, не передавая типа объекта деструктору, программист лишает объект данного типа полных преимуществ управления динамической памятью в Borland Pascal.

Деструкторы в действительности становятся самими собой тогда, когда должны очищаться полиморфические объекты и когда должна освобождаться занимаемая ими память.

Полиморфические объекты — это те объекты, которые были присвоены родительскому типу благодаря правилам совместимости расширенных типов Borland Pascal. Экземпляр объекта типа THourly, присвоенный переменной типа TEmployee, является примером полиморфического объекта. Эти правила также могут быть применены к объектам; указатель на THourly может свободно быть присвоен указателю на TEmployee, а указуемый этим указателем объект опять же будет полиморфическим объектом. Термин «полиморфический» является подходящим, так как код, обрабатывающий объект, «не знает» точно во время компиляции,

какой тип объекта ему придется в конце концов обработать. Единственное, что он знает, — это то, что этот объект принадлежит иерархии объектов, являющихся потомками указанного типа объекта.

Очевидно, что размеры типов объектов отличаются. Поэтому, когда наступает время очистки размещенного в динамической памяти полиморфического объекта, то как же Dispose узнает, сколько байт динамического пространства нужно освобождать? Во время компиляции из полиморфического объекта нельзя извлечь никакой информации относительно размера объекта.

Деструктор разрешает эту головоломку путем обращения к тому месту, где эта информация записана, — в ТВМ переменных реализаций. В каждой ТВМ типа объекта содержится размер в байтах данного типа объекта. Таблица виртуальных методов любого объекта доступна посредством скрытого параметра Self, посылаемого методу при вызове метода. Деструктор является всего лишь разновидностью метода, и поэтому, когда объект вызывает его, деструктор получает копию Self через стек. Таким образом, если объект является полиморфическим во время компиляции, он никогда не будет полиморфическим во время выполнения благодаря позднему связыванию.

Для выполнения этого освобождения памяти при позднем связывании деструктор нужно вызывать как часть расширенного синтаксиса процедуры Dispose:

```
Dispose(P, Done);
```

(Вызов деструктора вне процедуры Dispose вообще не выполняет никакого освобождения памяти.) Здесь происходит на самом деле то, что сборщик мусора объекта, на который указывает P, выполняется как обычный метод. Однако, как только последнее действие выполнено, деструктор ищет размер реализации своего типа в ТВМ и пересылает размер процедуре Dispose. Процедура Dispose завершает процесс путем удаления правильного числа байт пространства динамической памяти, которое (пространство) до этого относилось к P^. Число освобождаемых байт будет правильным независимо от того, указывал ли P на экземпляр типа TSalaried, или он указывал на один из дочерних типов типа TSalaried, например на TCommissioned.

Заметьте, что сам по себе метод деструктора может быть пуст и выполнять только эту функцию:

```
destructor AnObject.Done;
begin
end;
```

То, что делается полезного в этом деструкторе, не является достоянием его тела, однако при этом компилятором генерируется код эпилога в ответ на зарезервированное слово destructor. Это напоминает модуль, который ничего не экспортирует, но который осуществляет некоторые невидимые действия за счет выполнения своей секции инициализации перед стартом программы. Все действия происходят «за кулисами».

4. Виртуальные методы

Метод становится виртуальным, если за его объявлением в типе объекта стоит новое зарезервированное слово *virtual.* Если объявляется метод в родительском типе как virtual, то все методы с аналогичными именами в дочерних типах также должны объявляться виртуальными во избежание ошибки компилятора.

Ниже приведены объекты из примера платежной ведомости, должным образом виртуализированные:

```
type
 PEmployee = ^TEmployee;
 TEmployee = object
 Name, Title: string[25];
 Rate: Real;
constructor Init (AName, ATitle: String; ARate: Real);
 function GetPayAmount : Real; virtual;
 function GetName : String;
 function GetTitle : String;
 function GetRate : Real;
 procedure Show; virtual;
end;
 PHourly = ^THourly;
 THourly = object(TEmployee);
```

```
 Time: Integer;
constructor Init (AName, ATitle: String; ARate: Real; Time: Integer);
 function GetPayAmount : Real; virtual;
 function GetTime : Integer;
 end;

 PSalaried = ^TSalaried;
 TSalaried = object(TEmployee);
 function GetPayAmount : Real; virtual;
 end;

 PCommissioned = ^TCommissioned;
 TCommissioned = object(Salaried);
 Commission : Real;
 SalesAmount : Real;
 constructor Init (AName, ATitle: String; ARate,
                   ACommission, ASalesAmount: Real);
 function GetPayAmount : Real; virtual;
 end;
```

Конструктор является специальным типом процедуры, которая выполняет некоторую установочную работу для механизма виртуальных методов. Более того, конструктор должен вызываться перед вызовом любого виртуального метода. Вызов виртуального метода без предварительного вызова конструктора может привести к блокированию системы, а у компилятора нет способа проверить порядок вызова методов.

Каждый тип объекта, имеющий виртуальные методы, обязан иметь конструктор.

Предупреждение

Конструктор должен вызываться перед вызовом любого другого виртуального метода. Вызов виртуального метода без предыдущего обращения к конструктору может вызвать блокировку системы, и компилятор не сможет проверить порядок, в котором вызываются методы.

Примечание

Для конструкторов объекта предлагается использовать *идентификатор Init.*

Каждый отдельный экземпляр объекта должен инициализироваться отдельным вызовом конструктора. Недостаточно инициализировать один экземпляр объекта и затем присваивать этот экземпляр другим. Другие экземпляры, даже если они могут содержать правильные данные, не будут инициализированы оператором присваивания и заблокируют систему при любых вызовах их виртуальных методов. Например:

```
var
 FBee, GBee: Bee; { создать два экземпляра Bee }
begin
 FBee.Init(5, 9)  { вызов конструктора для FBee }
 GBee := FBee;  { Gbee недопустим! }
end;
```

Что же именно создает конструктор? Каждый тип объекта содержит нечто, называемое таблицей виртуального метода (ТВМ) в сегменте данных. ТВМ содержит размер типа объекта и для каждого виртуального метода указатель на код, выполняющий данный метод. Конструктор устанавливает связь между вызывающей его реализацией объекта и ТВМ типа объекта.

Важно помнить, что имеется только одна ТВМ для каждого типа объекта. Отдельные экземпляры типа объекта (т. е. переменные этого типа) содержат только соединение с ТВМ, но не саму ТВМ. Конструктор устанавливает значение этого соединения в ТВМ. Именно благодаря этому нигде нельзя запустить выполнение перед вызовом конструктора.

5. Поля данных объекта и формальные параметры метода

Выводом из того факта, что методы и их объекты разделяют общую область действия, является то, что формальные параметры метода не могут быть идентичными любому из полей данных объекта. Это является не каким-то новым ограничением, налагаемым объектно-ориентированным программированием, а, скорее, теми же самыми старыми правилами области действия, которые

Pascal имел всегда. Это то же самое, что и запрет для формальных параметров процедуры быть идентичными локальным переменным этой процедуры:

```
procedure CrunchIt(Crunchee: MyDataRec, Crunchby,
  ErrorCode: integer);
var
 A, B: char;
 ErrorCode: integer;
begin
 .
 .
 .
```

Локальные переменные процедуры и ее формальные параметры совместно используют общую область действия и поэтому не могут быть идентичными. Будет получено сообщение «Error 4: Duplicate identifier» (Ошибка 4; Повторение идентификатора), если попытаться компилировать что-либо подобное, та же ошибка возникает при попытке присвоить формальному параметру метода имени поля объекта, которому даннёый метод принадлежит.

Обстоятельства несколько отличаются, так как помещение заголовка процедуры внутрь структуры данных является намеком на новшество в Turbo Pascal, но основные принципы области действия Pascal не изменились.

ЛЕКЦИЯ № 13.
Совместимость типов объектов

1. Инкапсуляция

Объединение в объекте кода и данных называется инкапсуляцией. В принципе, возможно предоставить достаточное количество методов, благодаря которым пользователь объекта никогда не будет обращаться к полям объекта непосредственно. Некоторые другие объектно-ориентированные языки, например Smalltalk, требуют обязательной инкапсуляции, однако в Borland Pascal имеется выбор.

Например, объекты TEmployee и THourly написаны таким образом, что совершенно исключена необходимость прямого обращения к их внутренним полям данных:

```
type
 TEmployee = object
 Name, Title: string[25];
 Rate: Real;
 procedure Init (AName, ATitle: string; ARate: Real);
 function GetName : String;
 function GetTitle : String;
 function GetRate : Real;
 function GetPayAmount : Real;
end;

 THourly = object(TEmployee)
 Time: Integer;
 procedure Init(AName, ATitle: string; ARate:
                Real, Atime: Integer);
 function GetPayAmount : Real;
end;
```

Здесь присутствуют только четыре поля данных: Name, Title, Rate и Time. Методы GetName и GetTitle выводят фамилию работающего и его должность соответственно. Метод GetPayAmount

использует Rate, а в случае работающего THourly и Time для вычисления суммы выплат работающему. Здесь уже нет необходимости обращаться непосредственно к этим полям данных.

Предположив существование экземпляра AnHourly типа THourly, мы могли бы использовать набор методов для манипулирования полями данных AnHourly, например:

```
with AnHourly do
begin
 Init (Aleksandr Petrov, Fork lift operator' 12.95, 62);
 {Выводит на экран фамилию, должность и сумму выплат}
 Show;
end;
```

Следует обратить внимание, что доступ к полям объекта осуществляется не иначе, как только с помощью методов этого объекта.

2. Расширяющиеся объекты

К сожалению, стандартный Pascal не предоставляет никаких возможностей для создания гибких процедур, позволяющих работать с абсолютно разными типами данных. Объектно-ориентированное программирование решает эту проблему с помощью наследования: если определен порожденный тип, то методы порождающего типа наследуются, однако при желании они могут переопределяться. Для переопределения наследуемого метода попросту описывается новый метод с тем же именем, что и наследуе-мый метод, но с другим телом и (при необходимости) с другим множеством параметров.

Определим дочерний по отношению к TEmployee тип, представляющий работника, которому платится часовая ставка, в следующем примере:

```
const
PayPeriods = 26;  { периоды выплат }
OvertimeThreshold = 80; { на период выплаты }
OvertimeFactor = 1.5;  { почасовой коэффициент }

type
THourly = object(TEmployee)
Time: Integer;
```

```
procedure Init(AName, ATitle: string; ARate:
   Real, Atime: Integer);
function GetPayAmount : Real;
end;

procedure THourly.Init(AName, ATitle: string;
                           ARate: Real, Atime: Integer);
begin
TEmployee.Init(AName, ATitle, ARate);
Time := ATime;
end;

function THourly.GetPayAmount: Real;
var
Overtime: Integer;
begin
Overtime := Time – OvertimeThreshold;
if Overtime > 0 then
GetPayAmount := RoundPay(OvertimeThreshold * Rate +
                           Rate OverTime * OvertimeFactor * Rate)
else
GetPayAmount := RoundPay(Time * Rate)
end;
```

Человек, которому платится часовая ставка, является работающим: он обладает всем тем, что используется для определения объекта TEmployee (фамилией, должностью, ставкой), и лишь количество получаемых почасовиком денег зависит от того, сколько часов он отработал за период, подлежащий оплате. Таким образом, для THourly требуется еще и поле времени Time.

Так как THourly определяет новое поле Time, его инициализация требует нового метода Init, который инициализирует и время, и наследованные поля. Вместо того, чтобы непосредственно присвоить значения наследованным полям, таким как Name, Title и Rate, почему бы не использовать вновь метод инициализации объекта TEmployee (иллюстрируемый первым оператором THourly.Init).

Вызов метода, который переопределяется, не является лучшим стилем. В общем случае возможно, что TEmployee.Init выполняет важную, однако скрытую инициализацию.

Вызывая переопределяемый метод, необходимо быть уверенным в том, что порожденный тип объекта включает функциональность родителя. Кроме того, любое изменение в родительском методе автоматически оказывает влияние на все порожденные.

После вызова TEmployee.Init, THourly.Init может затем выполнить свою собственную инициализацию, которая в этом случае состоит только в присвоении значения, переданного в ATime.

Другим примером переопределяемого метода является функция THourly.GetPayAmount, вычисляющая сумму выплат работающему на почасовой ставке. В действительности, каждый тип объекта TEmployee имеет свой метод GetPayAmount, так как тип работающего зависит от того, как производится расчет. Метод THourly.GetPayAmount должен учитывать, сколько часов работал сотрудник, были ли сверхурочные работы, каков коэффициент увеличения за сверхурочные работы и т. д.

Метод TSalaried.GetPayAmount должен лишь делить ставку работающего на число выплат в каждом году (в нашем примере).

```
unit Workers;
interface

const
 PayPeriods = 26; {в год}
 OvertimeThreshold = 80; {за каждый период оплаты}
 OvertimeFactor =1.5; {увеличение против обычной оплаты}

type
 TEmployee = object
 Name, Title: string[25];
 Rate: Real;
 procedure Init (AName, ATitle: string; ARate: Real);
 function GetName : String;
 function GetTitle : String;
 function GetRate : Real;
 function GetPayAmount : Real;
end;

 THourly = object(TEmployee)
 Time: Integer;
 procedure Init(AName, ATitle: string; ARate:
                  Real, Atime: Integer);
```

```
 function GetPayAmount : Real;
 function GetTime : Real;
end;

 TSalaried = object(TEmployee)
 function GetPayAmount : Real;
end;

 TCommissioned = object(TSalaried)
 Commission : Real;
 SalesAmount : Real;

constructor Init (AName, ATitle: String; ARate,
                 ACommission, ASalesAmount: Real);
 function GetPayAmount : Real;
end;

implementation

function RoundPay(Wages: Real) : Real;
{округляем сумму выплат, чтобы игнорировать суммы меньше
денежной единицы}
begin
 RoundPay := Trunc(Wages * 100) / 100;
 .
 .
 .
```

TEmployee является вершиной нашей иерархии объектов и содержит первый метод GetPayAmount.

```
function TEmployee.GetPayAmount : Real;
begin
 RunError(211); { дать ошибку этапа выполнения }
end;
```

Может вызвать удивление тот факт, что метод дает ошибку времени выполнения. Если вызывается Employee.GetPayAmount, то в программе возникает ошибка. Почему? Потому что TEmployee является вершиной нашей иерархии объектов и не определяет реального рабочего; следовательно, ни один из методов TEmployee не

вызывается определенным образом, хотя они и могут быть наследованными. Все наши работники являются либо почасовиками, либо имеют оклады, либо работают на сдельщине. Ошибка на этапе выполнения прекращает выполнение программы и выводит 211, что соответствует сообщению об ошибке, связанной с вызовом абстрактного метода (если программа по ошибке вызывает TEmployee.GetPayAmount).

Ниже приводится метод THourly.GetPayAmount, в котором учитываются такие вещи, как сверхурочная оплата, число отработанных часов и т. д.

```
function THourly.GetPayAMount : Real;
var
 OverTime: Integer;
begin
 Overtime := Time – OvertimeThreshold;
 if Overtime > 0 then
 GetPayAmount := RoundPay(OvertimeThreshold * Rate +
                         Rate OverTime * OvertimeFactor * Rate)
 else
 GetPayAmount := RoundPay(Time * Rate)
end;
```

Метод TSalaried.GetPayAmount намного проще; в нем ставка делится на число выплат:

```
function TSalaried.GetPayAmount : Real;
begin
 GetPayAmount := RoundPay(Rate / PayPeriods);
end;
```

Если взглянуть на метод TCommissioned.GetPayAmount, то будет видно, что он вызывает TSalaried.GetPayAmount, вычисляет комиссионные и прибавляет их к величине, возвращаемой методом TSalaried.GetPayAmount.

```
function TCommissioned.GetPayAmount : Real;
begin
 GetPayAmount := RoundPay(TSalaried.GetPayAmount +
                              Commission * SalesAmount);
end;
```

Важное замечание: хотя методы могут быть переопределены, поля данных переопределяться не могут. После того как было определено поле данных в иерархии объекта, никакой дочерний тип не может определить поле данных в точности с таким же именем.

3. Совместимость типов объектов

Наследование до некоторой степени изменяет правила совместимости типов в Borland Pascal. Помимо всего прочего, порожденный тип наследует совместимость типов всех своих порождающих типов.

Эта расширенная совместимость типов принимает три формы:

1) между реализациями объектов;
2) между указателями на реализации объектов;
3) между формальными и фактическими параметрами.

Однако очень важно помнить, что во всех трех формах совместимость типов расширяется только от потомка к родителю. Другими словами, дочерние типы могут свободно использоваться вместо родительских, но не наоборот.

Например, TSalaried является потомком TEmployee, а TCommissioned — потомком TSalaried. Помня об этом, рассмотрим следующие описания:

```
type
 PEmployee = ^TEmployee;
 PSalaried = ^TSalaried;
 PCommissioned = ^TCommissioned;
var
 AnEmployee: TEmployee;
 ASalaried: TSalaried;
 PCommissioned: TCommissioned;
 TEmployeePtr: PEmployee;
 TSalariedPtr: PSalaried;
 TCommissionedPtr: PCommissioned;
```

При данных описаниях справедливы следующие операторы присваивания:

```
 AnEmployee :=ASalaried;
 ASalaried := ACommissioned;
 TCommissionedPtr := ACommissioned;
```

Примечание

Порождающему объекту можно присвоить экземпляр любого из его порожденных типов. Обратные присваивания недопустимы.

Эта концепция является новой для Pascal, и вначале, возможно, трудновато запомнить, в каком порядке следует совместимость типов. Необходимо думать следующим образом: источник должен быть в состоянии полностью заполнить приемник. Порожденные типы содержат все, что содержат их порождающие типы благодаря свойству наследования. Поэтому порожденный тип имеет либо в точности такой же размер, либо (что чаще всего и бывает) он больше своего родителя, но никогда не бывает меньше. Присвоение порождающего (родительского) объекта порожденному (дочернему) могло бы оставить некоторые поля порожденного объекта неопределенными, что опасно и поэтому недопустимо.

В операторах присваивания из источника в приемник будут копироваться только поля, являющиеся общими для обоих типов. В операторе присваивания:

```
AnEmployee := ACommissioned;
```

Только поля Name, Title и Rate из ACommissioned будут скопированы в AnEmployee, так как только эти поля являются общими для TCommissioned и TEmployee. Совместимость типов работает также между указателями на типы объектов и подчиняется тем же общим правилам, что и для реализаций объектов. Указатель на потомка может быть присвоен указателю на родителя. Если дать предыдущие определения, то следующие присваивания указателей будут допустимыми:

```
TSalariedPtr := TCommissionedPtr;
TEmployeePtr := TSalariedPtr;
TEmployeePtr := PCommissionedPtr;
```

Следует помнить, что обратные присваивания недопустимы!

Формальный параметр (либо значение, либо параметр-переменная) данного объектного типа может принимать в качестве фактического параметра объект своего же типа или объекты всех дочерних типов. Если определить заголовок процедуры следующим образом:

```
procedure CalcFedTax(Victim: TSalaried);
```

то допустимыми типами фактических параметров могут быть TSalaried или TCommissioned, но не тип TEmployee. Victim также может быть параметром-переменной. При этом выполняются те же правила совместимости.

Замечание

Между параметрами-значениями и параметрами-переменными есть коренное отличие. Параметр-значение является указателем на действительный, посылаемый в качестве параметра объект, тогда как параметр-переменная является только копией фактического параметра. Более того, эта копия включает только те поля, которые входят в тип формального параметра-значения. Это означает, что фактический параметр буквально преобразуется к типу формального параметра. Параметр-переменная больше напоминает приведение к образцу, в том смысле, что фактический параметр остается неизменным.

Аналогично, если формальный параметр является указателем на тип объекта, фактический параметр может быть указателем на этот тип объекта или на любой дочерний тип. Пусть дан заголовок процедуры:

```
procedure Worker.Add (AWorker: PSalaried);
```

Тогда допустимыми типами фактических параметров могут быть PSalaried или PCommissioned, но не тип PEmployee.

ЛЕКЦИЯ № 14. Ассемблер

1. Об ассемблере

Когда-то ассемблер был языком, без знания которого нельзя было заставить компьютер сделать что-либо полезное. Постепенно ситуация менялась. Появлялись более удобные средства общения с компьютером. Но в отличие от других языков ассемблер не умирал, более того, он не мог сделать этого в принципе. Почему? В поисках ответа попытаемся понять, что такое язык ассемблера вообще.

Если коротко, то язык ассемблера — это символическое представление машинного языка. Все процессы в машине на самом низком, аппаратном уровне приводятся в действие только командами (инструкциями) машинного языка. Отсюда понятно, что, несмотря на общее название, язык ассемблера для каждого типа компьютера свой. Это касается и внешнего вида программ, написанных на ассемблере, и идей, отражением которых этот язык является.

По-настоящему решить проблемы, связанные с аппаратурой (или, даже более того, зависящие от аппаратуры, как, к примеру, повышение быстродействия программы), невозможно без знания ассемблера.

Программист или любой другой пользователь могут использовать любые высокоуровневые средства вплоть до программ построения виртуальных миров и, возможно, даже не подозревать, что на самом деле компьютер выполняет не команды языка, на котором написана его программа, а их трансформированное представление в форме скучной и унылой последовательности команд совсем другого языка — машинного. А теперь представим, что у такого пользователя возникла нестандартная проблема. К примеру, его программа должна работать с некоторым необычным устройством или выполнять другие действия, требующие знания принципов работы аппаратуры компьютера. Каким бы хорошим ни был язык, на котором программист написал свою программу, без знания ассемблера ему не обойтись. И не случай-

но практически все компиляторы языков высокого уровня содержат средства связи своих модулей с модулями на ассемблере либо поддерживают выход на ассемблерный уровень программирования.

Компьютер составлен из нескольких физических устройств, каждое из которых подключено к одному блоку, называемому системным. Чтобы понять их функциональное назначение, посмотрим на структурную схему типичного компьютера (рис. 1). Она не претендует на безусловную точность и имеет целью лишь показать назначение, взаимосвязь и типовой состав элементов современного персонального компьютера.

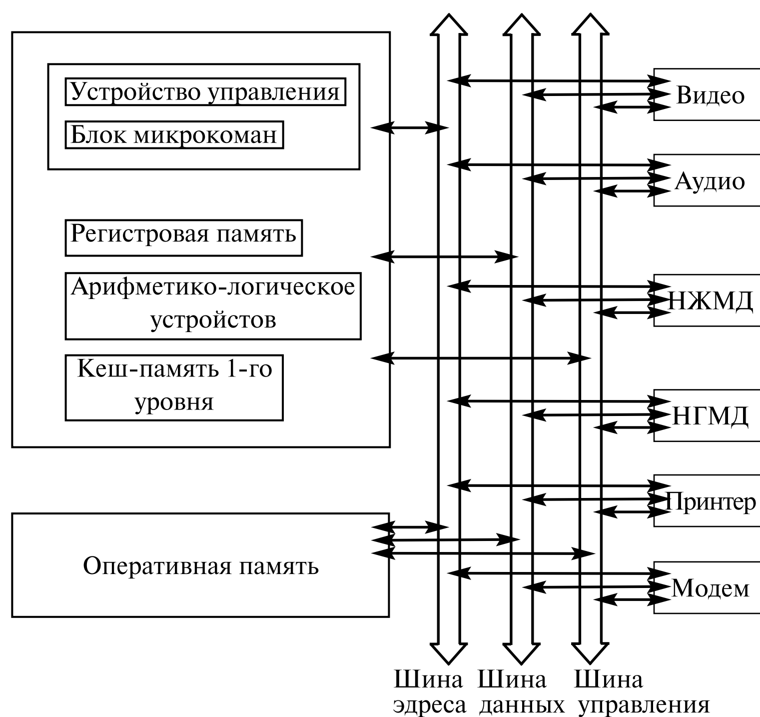


Рис. 1. Структурная схема персонального компьютера

2. Программная модель микропроцессора

На современном компьютерном рынке наблюдается большое разнообразие различных типов компьютеров. Поэтому возможно

предположить возникновение у потребителя вопроса о том, как оценить возможности конкретного типа (или модели) компьютера и его отличительные особенности от компьютеров других типов (моделей). Чтобы собрать воедино все понятия, характеризующие компьютер с точки зрения его функциональных программно-управляемых свойств, существует специальный термин — архитектура ЭВМ. Впервые понятие архитектура ЭВМ стало упоминаться с появлением машин 3-го поколения для их сравнительной оценки.

К изучению языка ассемблера любого компьютера имеет смысл приступать только после выяснения того, какая часть компьютера оставлена видимой и доступной для программирования на этом языке. Это так называемая программная модель компьютера, частью которой является программная модель микропроцессора, которая содержит тридцать два регистра, в той или иной мере доступных для использования программистом.

Данные регистры можно разделить на две большие группы:

1) 6 пользовательских регистров;
2) 16 системных регистров.

3. Пользовательские регистры

Как следует из названия, пользовательскими регистры называются потому, что программист может использовать их при написании своих программ. К этим регистрам относятся (рис. 2):

1) восемь 32-битных регистров, которые могут использоваться программистами для хранения данных и адресов (их еще называют регистрами общего назначения (РОН)):
   eax/ax/ah/al;
   ebx/bx/bh/bl;
   edx/dx/dh/dl;
   ecx/cx/ch/cl;
   ebp/bp;
   esi/si;
   edi/di;
   esp/sp.
2) шесть регистров сегментов: cs, ds, ss, es, fs, gs;
3) регистры состояния и управления:
   регистр флагов eflags/flags;
   регистр указателя команды eip/ip.

Регистры общего назначения:

Регистр	31	15	7	0
eax		ax		
		ah	al	
edx		dx		
		dh	dl	
ecx		cx		
		ch	cl	
ebx		bx		
		bh	bl	
ebp		bp		
esi		si		
edi		di		
esp		sp		

Сегментные регистры:

15	0
cs	
ss	
ds	
es	
fs	
qs	

Регистры флагов и указателя команд:

Регистр	31	15	0
eflags		ax	
eip		ip	

Рис. 2. Пользовательские регистры

Многие из этих регистров приведены с наклонной разделительной чертой. Это не разные регистры — это части одного большого 32-разрядного регистра. Их можно использовать в программе как отдельные объекты.

4. Регистры общего назначения

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

Перечислим регистры, относящиеся к группе регистров общего назначения. Так как эти регистры физически находятся в микропроцессоре внутри арифметико-логического устройства (АЛУ), то их еще называют регистрами АЛУ:

1) eax/ax/ah/al (Accumulator register) — аккумулятор.
Применяется для хранения промежуточных данных. В некоторых командах использование этого регистра обязательно;
2) ebx/bx/bh/bl (Base register) — базовый регистр.
Применяется для хранения базового адреса некоторого объекта в памяти;
3) ecx/cx/ch/cl (Count register) — регистр-счетчик.
Применяется в командах, производящих некоторые повторяющиеся действия. Его использование зачастую неявно и скрыто в алгоритме работы соответствующей команды.
К примеру, команда организации цикла loop, кроме передачи управления команде, находящейся по некоторому адресу, анализирует и уменьшает на единицу значение регистра ecx/cx;
4) edx/dx/dh/dl (Data register) — регистр данных.
Так же, как и регистр eax/ax/ah/al, он хранит промежуточные данные. В некоторых командах его использование обязательно; для некоторых команд это происходит неявно.

Следующие два регистра используются для поддержки так называемых цепочечных операций, т. е. операций, производящих последовательную обработку цепочек элементов, каждый из которых может иметь длину 32, 16 или 8 бит:

1) esi/si (Source Index register) — индекс источника.

Этот регистр в цепочечных операциях содержит текущий адрес элемента в цепочке-источнике;

2) edi/di (Destination Index register) — индекс приемника (получателя). Этот регистр в цепочечных операциях содержит текущий адрес в цепочке-приемнике.

В архитектуре микропроцессора на программно-аппаратном уровне поддерживается такая структура данных, как стек. Для работы со стеком в системе команд микропроцессора есть специальные команды, а в программной модели микропроцессора для этого существуют специальные регистры:

1) esp/sp (Stack Pointer register) — регистр указателя стека. Содержит указатель вершины стека в текущем сегменте стека.

2) ebp/bp (Base Pointer register) — регистр указателя базы кадра стека. Предназначен для организации произвольного доступа к данным внутри стека.

Использование жесткого закрепления регистров для некоторых команд позволяет более компактно кодировать их машинное представление. Знание этих особенностей позволит при необходимости хотя бы на несколько байт сэкономить память, занимаемую кодом программы.

5. Сегментные регистры

В программной модели микропроцессора имеется шесть сегментных регистров: cs, ss, ds, es, gs, fs.

Их существование обусловлено спецификой организации и использования оперативной памяти микропроцессорами Intel. Она заключается в том, что микропроцессор аппаратно поддерживает структурную организацию программы в виде трех частей, называемых сегментами. Соответственно такая организация памяти называется сегментной.

Для того чтобы указать на сегменты, к которым программа имеет доступ в конкретный момент времени, и предназначены сегментные регистры. Фактически (с небольшой поправкой) в этих регистрах содержатся адреса памяти, с которых начинаются соответствующие сегменты. Логика обработки машинной команды построена так, что при выборке команды, доступе к данным программы или к стеку неявно используются адреса во вполне определенных сегментных регистрах.

Микропроцессор поддерживает следующие типы сегментов.

1. Сегмент кода. Содержит команды программы. Для доступа к этому сегменту служит *регистр cs* (code segment register) — сегментный регистр кода. Он содержит адрес сегмента с машинными командами, к которому имеет доступ микропроцессор (т. е. эти команды загружаются в конвейер микропроцессора).

2. Сегмент данных. Содержит обрабатываемые программой данные. Для доступа к этому сегменту служит *регистр ds* (data segment register) — сегментный регистр данных, который хранит адрес сегмента данных текущей программы.

3. Сегмент стека. Этот сегмент представляет собой область памяти, называемую стеком. Работу со стеком микропроцессор организует по следующему принципу: последний записанный в эту область элемент выбирается первым. Для доступа к этому сегменту служит *регистр ss* (stack segment register) — сегментный регистр стека, содержащий адрес сегмента стека.

4. Дополнительный сегмент данных. Не явно алгоритмы выполнения большинства машинных команд предполагают, что обрабатываемые ими данные расположены в сегменте данных, адрес которого находится в сегментном *регистре ds*. Если программе недостаточно одного сегмента данных, то она имеет возможность использовать еще три дополнительных сегмента данных. Но в отличие от основного сегмента данных, адрес которого содержится в сегментном *регистре ds*, при использовании дополнительных сегментов данных их адреса требуется указывать явно с помощью специальных префиксов переопределения сегментов в команде. Адреса дополнительных сегментов данных должны содержаться в *регистрах es, gs, fs* (extension data segment registers).

6. Регистры состояния и управления

В микропроцессор включены несколько регистров, которые постоянно содержат информацию о состоянии как самого микропроцессора, так и программы, команды которой в данный момент загружены на конвейер. К этим регистрам относятся:

1) регистр флагов eflags/flags;
2) регистр указателя команды eip/ip.

Используя эти регистры, можно получать информацию о результатах выполнения команд и влиять на состояние самого микропроцессора. Рассмотрим подробнее назначение и содержимое этих регистров

1. eflags/flags (flag register) — регистр флагов. Разрядность eflags/flags — 32/16 бит. Отдельные биты данного регистра имеют определенное функциональное назначение и называются флагами. Младшая часть этого регистра полностью аналогична регистру flags для i8086. На рисунке 3 показано содержимое регистра eflags.

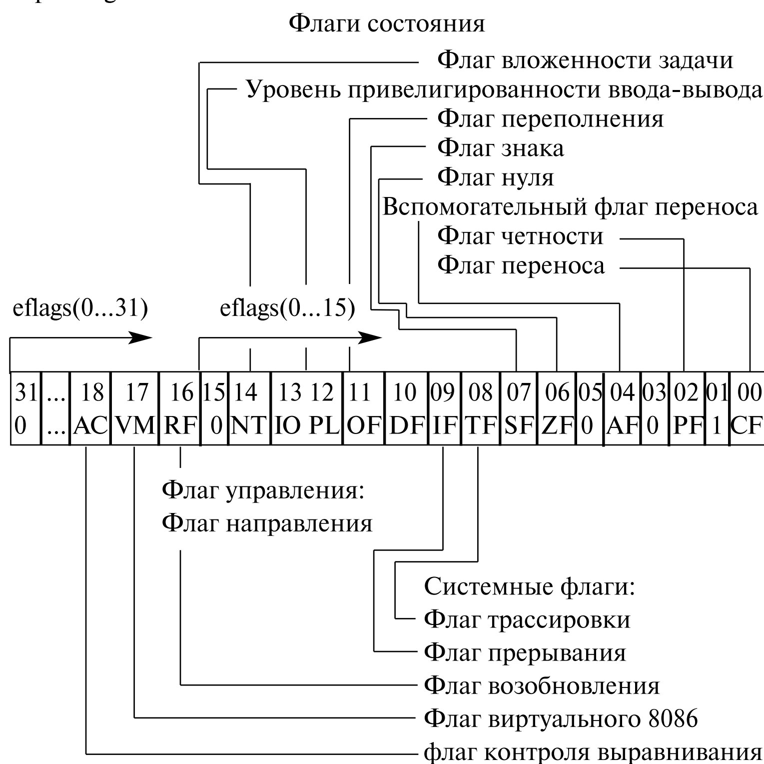


Рис. 3. Содержимое регистра eflags

Исходя из особенностей использования флаги регистра eflags/flags можно разделить на три группы:

1) восемь флагов состояния.

Эти флаги могут изменяться после выполнения машинных команд. Флаги состояния *регистра eflags* отражают особенно-

сти результата исполнения арифметических или логических операций. Это дает возможность анализировать состояние вычислительного процесса и реагировать на него с помощью команд условных переходов и вызовов подпрограмм. В таблице 1 приведены флаги состояния и указано их назначение.

2) один флаг управления.

Обозначается df (Directory Flag). Он находится в 10-м бите регистра eflags и используется цепочечными командами. Значение флага df определяет направление поэлементной обработки в этих операциях: от начала строки к концу (df = 0) либо наоборот, от конца строки к ее началу (df = 1).

Для работы с флагом df существуют специальные команды: cld (снять флаг df) и std (установить флаг df).

Применение этих команд позволяет привести флаг df в соответствие с алгоритмом и обеспечить автоматическое увеличение или уменьшение счетчиков при выполнении операций со строками.

3) пять системных флагов.

Управляют вводом-выводом, маскируемыми прерываниями, отладкой, переключением между задачами и виртуальным режимом 8086. Прикладным программам не рекомендуется модифицировать без необходимости эти флаги, так как в большинстве случаев это приведет к прерыванию работы программы. В таблице 2 перечислены системные флаги, их назначение.

Таблица 1

Флаги состояния

Мнемоника флага	Флаг	Номер бита в eflags	Содержание и назначение
Cf	Флаг переноса (Carry Flag)	0	1 — арифметическая операция произвела перенос из старшего бита результата. Старшим является 7, 15 или 31-й бит в зависимости от размерности операнда; 0 — переноса не было
Pf	Флаг паритета (Parity Flag)	2	1 — 8 младших разрядов (этот флаг — только для 8 младших разрядов операнда любого размера) результата содержат четное число единиц; 0 — 8 младших разрядов результата содержат нечетное число единиц

Продолжение таб. 1

Мнемоника флага	**Флаг**	**Номер бита в eflags**	**Содержание и назначение**
Af	Вспомогательный флаг переноса (Auxiliary carry Flag)	4	Только для команд, работающих с BCD-числами. Фиксирует факт заема из младшей тетрады результата: 1 — в результате операции сложения был произведен перенос из разряда 3 в старший разряд, или при вычитании был заем в разряд 3 младшей тетрады из значения в старшей тетраде;
Zf	Флаг нуля (Zero Flag)	6	1 — результат нулевой; 0 — результат ненулевой
Sf	Флаг знака (Sign Flag)	7	Отражает состояние старшего бита результата (биты 7, 15 или 31 для 8-, 16- или 32-разрядных операндов соответственно): 1 — старший бит результата равен 1; 0 — старший бит результата равен 0
Of	Флаг переполнения (Overflow Flag)	11	Флаг of используется для фиксирования факта потери значащего бита при арифметических операциях: 1 — в результате операции происходит перенос (заем) в (из) старшего, знакового бита результата (биты 7, 15 или 31 для 8-, 16 или 32-разрядных операндов соответственно); 0 —

Окончание табл. 1

Мнемоника флага	Флаг	Номер бита в eflags	Содержание и назначение
			в результате операции не происходит переноса (заема) в(из) старшего, знакового бита результата
Iopl	Уровень	12. 13	Используется в защищенном режиме работы

Системные флаги *Таблица 2*

Мнемоника флага	Флаг	Номер бита в eflags	Содержание и назначение
Tf	Флаг трассировки (Trace Flag)	8	Предназначен для организации пошаговой работы микропроцессора. 1 — микропроцессор генерирует прерывание с номером 1 после выполнения каждой машинной команды. Может использоваться при отладке программ, в частности отладчиками; 0 — обычная работа
If	Флаг прерывания (Interrupt enable Flag)	9	Предназначен для разрешения или запрещения (маскирования) аппаратных прерываний (прерываний по входу INTR). 1 — аппаратные прерывания разрешены; 0 — аппаратные прерывания запрещены
Rf		16	Используется при обработке прерываний от регистров отладки.

Окончание табл. 2

Мнемоника флага	Флаг	Номер бита в eflags	Содержание и назначение
Vm		17	Признак работы микропроцессора в режиме виртуального 8086. 1 — процессор работает в режиме виртуального 8086; 0 — процессор работает в реальном или защищенном режиме
Ac	Флаг контроля выравнивания (Alignment Check)	18	Предназначен для разрешения контроля выравнивания при обращениях к памяти. Используется совместно с битом am в системном регистре cr0. К примеру, Pentium разрешает размещать команды и данные с любого адреса. Если требуется контролировать выравнивание данных и команд по адресам, кратным 2 или 4, то установка данных битов приведет к тому, что все обращения по некратным адресам будут возбуждать исключительную ситуацию

2. eip/ip (Instraction Pointer register) — регистр-указатель команд. Регистр eip/ip имеет разрядность 32 / 16 бит и содержит смещение следующей подлежащей выполнению команды относительно содержимого сегментного регистра cs в текущем сегменте команд. Этот регистр непосредственно недоступен программисту, но загрузка и изменение его значения производятся различными командами управления, к которым относятся коман-ды условных и безусловных переходов, вызова процедур и возврата из процедур. Возникновение прерываний также приводит к модификации регистра eip/ip .

ЛЕКЦИЯ № 15. Регистры

1. Системные регистры микропроцессора

Само название этих регистров говорит о том, что они выполняют специфические функции в системе. Использование системных регистров жестко регламентировано. Именно они обеспечивают работу защищенного режима. Их также можно рассматривать как часть архитектуры микропроцессора, которая намеренно оставлена видимой для того, чтобы квалифицированный системный программист мог выполнить самые низкоуровневые операции.

Системные регистры можно разделить на три группы:

1) четыре регистра управления;
2) четыре регистра системных адресов;
3) восемь регистров отладки.

2. Регистры управления

В группу регистров управления входят четыре регистра: cr0, cr1, cr2, cr3. Эти регистры предназначены для общего управления системой. Регистры управления доступны только программам с уровнем привилегий 0.

Хотя микропроцессор имеет четыре регистра управления, доступными являются только три из них — исключается cr1, функции которого пока не определены (он зарезервирован для будущего использования).

Регистр cr0 содержит системные флаги, управляющие режимами работы микропроцессора и отражающие его состояние глобально, независимо от конкретных выполняющихся задач.

Назначение системных флагов:

1) pe (Protect Enable), бит 0 — разрешение защищенного режима работы. Состояние этого флага показывает, в каком из двух режимов — реальном (pe = 0) или защищенном (pe = 1) — работает микропроцессор в данный момент времени;
2) mp (Math Present), бит 1 — наличие сопроцессора. Всегда 1;

3) ts (Task Switched), бит 3 — переключение задач.

Процессор автоматически устанавливает этот бит при переключении на выполнение другой задачи;

4) am (Alignment Mask), бит 18 — маска выравнивания.

Этот бит разрешает (am = 1) или запрещает (am = 0) контроль выравнивания;

5) cd (Cache Disable), бит 30 — запрещение кеш-памяти.

С помощью этого бита можно запретить (cd = 1) или разрешить (cd = 0) использование внутренней кеш-памяти (кеш-памяти первого уровня);

6) pg (PaGing), бит 31 — разрешение (pg = 1) или запрещение (pg = 0) страничного преобразования.

Флаг используется при страничной модели организации памяти.

Регистр cr2 используется при страничной организации оперативной памяти для регистрации ситуации, когда текущая команда обратилась по адресу, содержащемуся в странице памяти, отсутствующей в данный момент времени в памяти.

В такой ситуации в микропроцессоре возникает исключительная ситуация с номером 14, и линейный 32-битный адрес команды, вызвавшей это исключение, записывается в регистр cr2. Имея эту информацию, обработчик исключения 14 определяет нужную страницу, осуществляет ее подкачку в память и возобновляет нормальную работу программы;

Регистр cr3 также используется при страничной организации памяти. Это так называемый регистр каталога страниц первого уровня. Он содержит 20-битный физический базовый адрес каталога страниц текущей задачи. Этот каталог содержит 1024 32-битных дескриптора, каждый из которых содержит адрес таблицы страниц второго уровня. В свою очередь, каждая из таблиц страниц второго уровня содержит 1024 32-битных дескриптора, адресующих страничные кадры в памяти. Размер страничного кадра — 4 Кбайта.

3. Регистры системных адресов

Эти регистры еще называют регистрами управления памятью.

Они предназначены для защиты программ и данных в мультизадачном режиме работы микропроцессора. При работе в защищенном режиме микропроцессора адресное пространство делится на:

1) глобальное — общее для всех задач;

2) локальное — отдельное для каждой задачи.

Этим разделением и объясняется присутствие в архитектуре микропроцессора следующих системных регистров:

1) регистра таблицы глобальных дескрипторов gdtr (Global Descriptor Table Register), имеющего размер 48 бит и содержащего 32-битовый (биты 16–47) базовый адрес глобальной дескрипторной таблицы GDT и 16-битовое (биты 0–15) значение предела, представляющее собой размер в байтах таблицы GDT;

2) регистра таблицы локальных дескрипторов ldtr (Local Descriptor Table Register), имеющего размер 16 бит и содержащего так называемый селектор дескриптора локальной дескрипторной таблицы LDT. Этот селектор является указателем в таблице GDT, который и описывает сегмент, содержащий локальную дескрипторную таблицу LDT;

3) регистра таблицы дескрипторов прерываний idtr (Interrupt Descriptor Table Register), имеющего размер 48 бит и содержащего 32-битовый (биты 16—47) базовый адрес дескрипторной таблицы прерываний IDT и 16-битовое (биты 0–15) значение предела, представляющее собой размер в байтах таблицы IDT;

4) 16-битового регистра задачи tr (Task Register), который подобно регистру ldtr, содержит селектор, т. е. указатель на дескриптор в таблице GDT. Этот дескриптор описывает текущий сегмент состояния задачи (TSS — Task Segment Status). Этот сегмент создается для каждой задачи в системе, имеет жестко регламентированную структуру и содержит контекст (текущее состояние) задачи. Основное назначение сегментов TSS — сохранять текущее состояние задачи в момент переключения на другую задачу.

4. Регистры отладки

Это очень интересная группа регистров, предназначенных для аппаратной отладки. Средства аппаратной отладки впервые появились в микропроцессоре i486. Аппаратно микропроцессор содержит восемь регистров отладки, но реально из них используются только шесть.

Регистры dr0, dr1, dr2, dr3 имеют разрядность 32 бита и предназначены для задания линейных адресов четырех точек прерывания. Используемый при этом механизм следующий: любой формируемый текущей программой адрес сравнивается с адресами в регистрах dr0 ... dr3 , и при совпадении генерируется исключение отладки с номером 1.

Регистр dr6 называется регистром состояния отладки. Биты этого регистра устанавливаются в соответствии с причинами, которые вызвали возникновение последнего исключения с номером 1.

Перечислим эти биты и их назначение:

1) b0 — если этот бит установлен в 1, то последнее исключение (прерывание) возникло в результате достижения контрольной точки, определенной в регистре dr0;

2) b1 — аналогично b0, но для контрольной точки в регистре dr1;

3) b2 — аналогично b0, но для контрольной точки в регистре dr2;

4) b3 — аналогично b0, но для контрольной точки в регистре dr3;

5) bd (бит 13) — служит для защиты регистров отладки;

6) bs (бит 14) — устанавливается в 1, если исключение 1 было вызвано состоянием флага tf = 1 в регистре eflags;

7) bt (бит 15) устанавливается в 1, если исключение 1 было вызвано переключением на задачу с установленным битом ловушки в TSS t = 1.

Все остальные биты в этом регистре заполняются нулями. Обработчик исключения 1 по содержимому dr6 должен определить причину, по которой произошло исключение, и выполнить необходимые действия.

Регистр dr7 называется регистром управления отладкой. В нем для каждого из четырех регистров контрольных точек отладки имеются поля, позволяющие уточнить следующие условия, при которых следует сгенерировать прерывание:

1) место регистрации контрольной точки — только в текущей задаче или в любой задаче. Эти биты занимают младшие 8 бит регистра dr7 (по 2 бита на каждую контрольную точку (фактически точку прерывания), задаваемую *регистрами dr0, dr1, dr2, dr3* соответственно).

Первый бит из каждой пары — это так называемое локальное разрешение; его установка говорит о том, что точка прерывания действует, если она находится в пределах адресного пространства текущей задачи.

Второй бит в каждой паре определяет глобальное разрешение, которое говорит о том, что данная контрольная точка действует в пределах адресных пространств всех задач, находящихся в системе;

2) тип доступа, по которому инициируется прерывание: только при выборке команды, при записи или при записи / чтении данных. Биты, определяющие подобную природу возникновения прерывания, локализуются в старшей части данного регистра.

Большинство из системных регистров программно доступно.

ЛЕКЦИЯ № 16. Программы на Ассемблере

1. Структура программы на ассемблере

Программа на ассемблере представляет собой совокупность блоков памяти, называемых сегментами памяти. Программа может состоять из одного или нескольких таких блоков-сегментов. Каждый сегмент содержит совокупность предложений языка, каждое из которых занимает отдельную строку кода программы.

Предложения ассемблера бывают четырех типов:

1) команды или инструкции, представляющие собой символические аналоги машинных команд. В процессе трансляции инструкции ассемблера преобразуются в соответствующие команды системы команд микропроцессора;

2) макрокоманды. Это оформляемые определенным образом предложения текста программы, замещаемые во время трансляции другими предложениями;

3) директивы, являющиеся указанием транслятору ассемблера на выполнение некоторых действий. У директив нет аналогов в машинном представлении;

4) строки комментариев, содержащие любые символы, в том числе и буквы русского алфавита. Комментарии игнорируются транслятором.

2. Синтаксис ассемблера

Предложения, составляющие программу, могут представлять собой синтаксическую конструкцию, соответствующую команде, макрокоманде, директиве или комментарию. Для того чтобы транслятор ассемблера мог распознать их, они должны формироваться по определенным синтаксическим правилам. Для этого лучше всего использовать формальное описание синтаксиса языка наподобие правил грамматики. Наиболее распространенные способы подобного описания языка программирования — синтаксические диаграммы и расширенные формы Бэкуса-Наура. Для практического использования более удобны синтаксические диа-

граммы. К примеру, синтаксис предложений ассемблера можно описать с помощью синтаксических диаграмм, показанных на следующих рисунках.

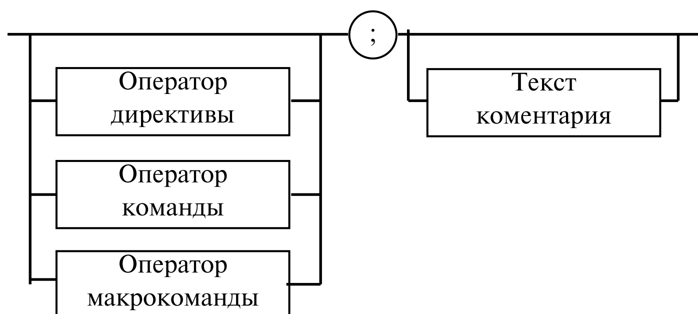


Рис. 4. Формат предложения ассемблера

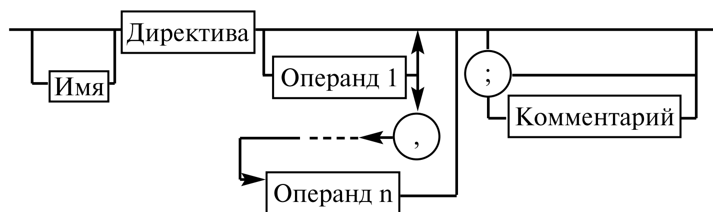


Рис. 5. Формат директив

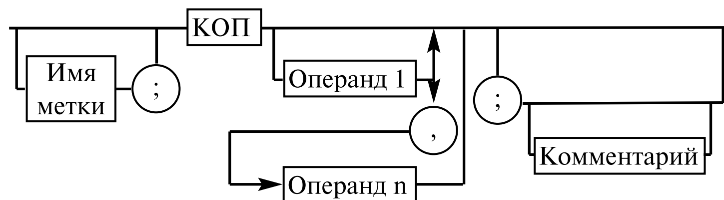


Рис. 6. Формат команд и макрокоманд

На этих рисунках:

1) имя метки — идентификатор, значением которого является адрес первого байта того предложения исходного текста программы, которое он обозначает;

2) имя — идентификатор, отличающий данную директиву от других одноименных директив. В результате обработки ассемблером определенной директивы этому имени могут быть присвоены определенные характеристики;

3) код операции (КОП) и директива — это мнемонические обозначения соответствующей машинной команды, макрокоманды или директивы транслятора;

4) операнды — части команды, макрокоманды или директивы ассемблера, обозначающие объекты, над которыми производятся действия. Операнды ассемблера описываются выражениями с числовыми и текстовыми константами, метками и идентификаторами переменных с использованием знаков операций и некоторых зарезервированных слов.

Как использовать синтаксические диаграммы? Очень просто: для этого нужно всего лишь найти и затем пройти путь от входа диаграммы (слева) к ее выходу (направо). Если такой путь существует, то предложение или конструкция синтаксически правильны. Если такого пути нет, значит, эту конструкцию компилятор не примет. При работе с синтаксическими диаграммами обращайте внимание на направление обхода, указываемое стрелками, так как среди путей могут быть и такие, по которым можно идти справа налево. По сути, синтаксические диаграммы отражают логику работы транслятора при разборе входных предложений программы.

Допустимыми символами при написании текста программ являются:

1) все латинские буквы: A – Z , a – z. При этом заглавные и строчные буквы считаются эквивалентными;
2) цифры от 0 до 9;
3) знаки ?, @ , $, _ , &;
4) разделители.

Предложения ассемблера формируются из лексем, представляющих собой синтаксически неразделимые последовательности допустимых символов языка, имеющие смысл для транслятора.

Лексемами являются следующие.

1. Идентификаторы — последовательности допустимых символов, использующиеся для обозначения таких объектов программы, как коды операций, имена переменных и названия меток. Правило записи идентификаторов заключается в следующем: идентификатор может состоять из одного или нескольких символов. В качестве символов можно использовать буквы латинского алфавита, цифры и некоторые специальные знаки — _, ?, $, @. Идентификатор не мо-

жет начинаться символом цифры. Длина идентификатора может быть до 255 символов, хотя транслятор воспринимает лишь первые 32, а остальные игнорирует. Регулировать длину возможных идентификаторов можно с использованием опции командной строки mv. Кроме этого, существует возможность указать транслятору на то, чтобы он различал прописные и строчные буквы либо игнорировал их различие (что и делается по умолчанию). Для этого применяются опции командной строки /mu, /ml, /mx.

2. Цепочки символов — последовательности символов, заключенные в одинарные или двойные кавычки.

3. Целые числа в одной из следующих систем счисления: двоичной, десятичной, шестнадцатеричной. Отождествление чисел при записи их в программах на ассемблере производится по определенным правилам:

1) десятичные числа не требуют для своего отождествления указания каких-либо дополнительных символов, например 25 или 139;

2) для отождествления в исходном тексте программы двоичных чисел необходимо после записи нулей и единиц, входящих в их состав, поставить латинское «b», например 10010101 b;

3) Шестнадцатеричные числа имеют больше условностей при своей записи:

   а) во-первых, они состоят из цифр 0...9, строчных и прописных букв латинского алфавита a, b, c, d, e, f или A, B, C, D, E, F.

   б) во-вторых, у транслятора могут возникнуть трудности с распознаванием шестнадцатеричных чисел из-за того, что они могут состоять как из одних цифр 0...9 (например, 190845), так и начинаться с буквы латинского алфавита (например, ef15). Для того, чтобы «объяснить» транслятору, что данная лексема не является десятичным числом или идентификатором, программист должен специальным образом выделять шестнадцатеричное число. Для этого на конце последовательности шестнадцатеричных цифр, составляющих шестнадцатеричное число, записывают латинскую букву «h». Это обязательное условие. Если шестнадцатеричное число начинается с буквы, то перед ним записывается ведущий нуль: 0 ef15 h.

Таким образом, мы разобрались с тем, как конструируются предложения программы ассемблера. Но это лишь самый поверхностный взгляд.

Практически каждое предложение содержит описание объекта, над которым или при помощи которого выполняется некоторое действие. Эти объекты называются операндами. Их можно определить так: операнды — это объекты (некоторые значения, регистры или ячейки памяти), на которые действуют инструкции или директивы, либо это объекты, которые определяют или уточняют действие инструкций или директив.

Операнды могут комбинироваться с арифметическими, логическими, побитовыми и атрибутивными операторами для расчета некоторого значения или определения ячейки памяти, на которую будет воздействовать данная команда или директива.

Рассмотрим подробнее характеристику операндов в нижеприведенной классификации:

1) постоянные или непосредственные операнды — число, строка, имя или выражение, имеющие некоторое фиксированное значение. Имя не должно быть перемещаемым, т. е. зависеть от адреса загрузки программы в память. К примеру, оно может быть определено операторами equ или =;

2) адресные операнды, задают физическое расположение операнда в памяти с помощью указания двух составляющих адреса: сегмента и смещения (рис. 7);

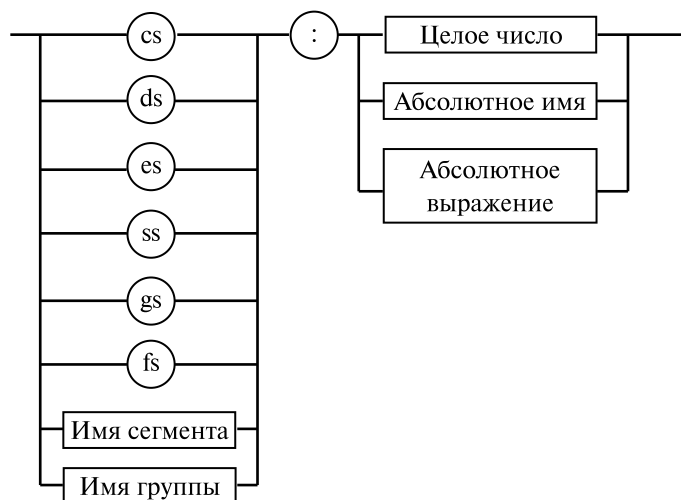


Рис. 7. Синтаксис описания адресных операндов

3) перемещаемые операнды — любые символьные имена, представляющие некоторые адреса памяти. Эти адреса могут обоз-

начать местоположение в памяти некоторых инструкций (если операнд — метка) или данных (если операнд — имя области памяти в сегменте данных).

Перемещаемые операнды отличаются от адресных тем, что они не привязаны к конкретному адресу физической памяти. Сегментная составляющая адреса перемещаемого операнда неизвестна и будет определена после загрузки программы в память для выполнения.

Счетчик адреса — специфический вид операнда. Он обозначается знаком $. Специфика этого операнда в том, что когда транслятор ассемблера встречает в исходной программе этот символ, то он подставляет вместо него текущее значение счетчика адреса. Значение счетчика адреса или, как его иногда называют, счетчика размещения представляет собой смещение текущей машинной команды относительно начала сегмента кода.

В формате листинга счетчику адреса соответствует вторая или третья колонка (в зависимости от того, присутствует или нет в листин-ге колонка с уровнем вложенности). Если взять в качестве примера любой листинг, то видно, что при обработке транслятором очередной команды ассемблера счетчик адреса увеличивается на длину сформированной машинной команды. Важно правильно понимать этот момент.

К примеру, обработка директив ассемблера не влечет за собой изменения счетчика. Директивы, в отличие от команд ассемблера, — это лишь указания транслятору на выполнение определенных действий по формированию машинного представления программы, и для них транслятором не генерируется никаких конструкций в памяти.

При использовании подобного выражения для перехода не забывайте о длине самой команды, в которой это выражение используется, так как значение счетчика адреса соответствует смещению в сегменте команд данной, а не следующей за ней команды. В нашем примере команда jmp занимает 2 байта. Но будьте осторожны, длина команды зависит от того, какие в ней используются операнды. Команда с регистровыми операндами будет короче команды, один из операндов которой расположен в память. В большинстве случаев эту информацию можно получить, зная формат машинной команды и анализируя колонку листинга с объектным кодом команды;

4) регистровый операнд — это просто имя регистра. В программе на ассемблере можно использовать имена всех регистров общего назначения и большинства системных регистров;

5) базовый и индексный операнды. Этот тип операндов используется для реализации косвенной базовой, косвенной индексной адресации или их комбинаций и расширений;

6) структурные операнды используются для доступа к конкретному элементу сложного типа данных, называемого структурой.

Записи (аналогично структурному типу) используются для доступа к битовому полю некоторой записи.

Операнды являются элементарными компонентами, из которых формируется часть машинной команды, обозначающая объекты, над которыми выполняется операция. В более общем случае операнды могут входить как составные части в более сложные образования, называемые выражениями. Выражения представляют собой комбинации операндов и операторов, рассматриваемые как единое целое. Результатом вычисления выражения может быть адрес некоторой ячейки памяти или некоторое константное (абсолютное) значение.

Возможные типы операндов мы уже рассмотрели. Перечислим теперь возможные типы операторов ассемблера и синтаксические правила формирования выражений ассемблера, и дадим краткую характеристику операторов.

1. Арифметические операторы. К ним относятся:

1) унарные « + » и « – »;
2) бинарные « + » и « – »;
3) умножения « * »;
4) целочисленного деления « / »;
5) получения остатка от деления « mod ».

Эти операторы расположены на уровнях приоритета 6, 7, 8 в таблице 4.

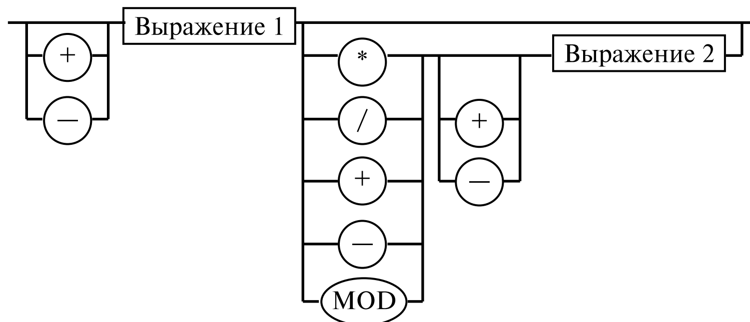


Рис. 8. Синтаксис арифметических операций

2. Операторы сдвига выполняют сдвиг выражения на указанное количество разрядов (рис. 9).

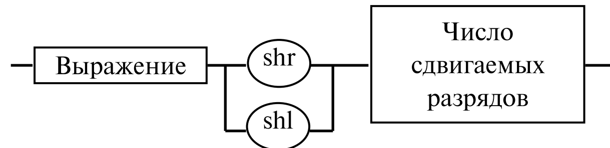


Рис. 9. Синтаксис операторов сдвига

3. Операторы сравнения (возвращают значение «истина» или «ложь») предназначены для формирования логических выражений (рис. 10 и табл. 3). Логическое значение «истина» соответствует цифровой единице, а «ложь» — нулю.

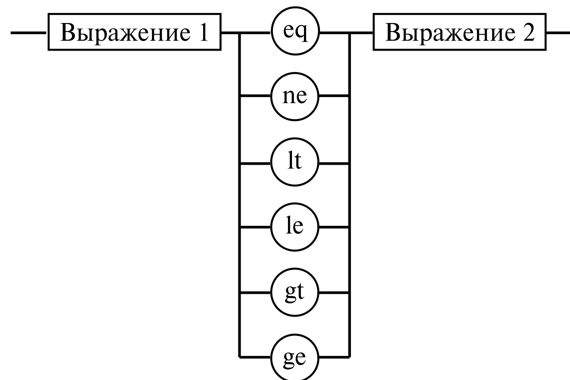


Рис. 10. Синтаксис операторов сравнения

Таблица 3

Операторы сравнения

<table>
<tr><th>Оператор</th><th>Значение</th></tr>
<tr><td>eq</td><td>ИСТИНА, если выражение_1 равно выражение_2</td></tr>
<tr><td>ne</td><td rowspan="2">ИСТИНА, если выражение_1 не равно выражение_2</td></tr>
<tr><td>lt</td></tr>
<tr><td>le</td><td>ИСТИНА, если выражение_1 меньше или равно выражение_2</td></tr>
<tr><td>gt</td><td>ИСТИНА, если выражение_1 больше выражение_2</td></tr>
<tr><td>ge</td><td>ИСТИНА, если выражение_1 больше или равно выражение_2</td></tr>
</table>

4. Логические операторы выполняют над выражениями побитовые операции (рис. 11). Выражения должны быть абсолютными, т. е. такими, численное значение которых может быть вычислено транслятором.

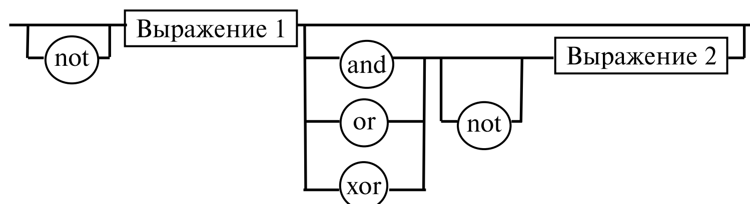


Рис. 11. Синтаксис логических операторов

5. Индексный оператор [ ]. Скобки тоже являются оператором, и транслятор их наличие воспринимает как указание сложить значение выражение_1 за этими скобками с выражение_2, заключенным в скобки (рис. 12).

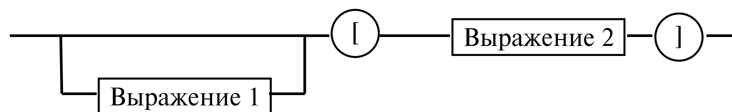


Рис. 12. Синтаксис индексного оператора

Заметим, что в литературе по ассемблеру принято следующее обозначение: когда в тексте речь идет о содержимом регистра, то его название берут в круглые скобки. Мы также будем придерживаться этого обозначения.

6. Оператор переопределения типа ptr применяется для переопределения или уточнения типа метки или переменной, определяемых выражением (рис. 13).

Тип может принимать одно из следующих значений: byte, word, dword, qword, tbyte, near, far .

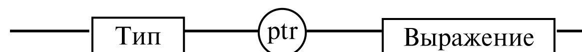


Рис. 13. Синтаксис оператора переопределения типа

7. Оператор переопределения сегмента «:» (двоеточие) заставляет вычислять физический адрес относительно конкретно задаваемой сегментной составляющей: «имя сегментного регистра»,

«имя сегмента» из соответствующей директивы SEGMENT или «имя группы» (рис. 14). При обсуждении сегментации мы говорили о том, что микропроцессор на аппаратном уровне поддерживает три типа сегментов — кода, стека и данных. В чем заключается такая аппаратная поддержка? К примеру, для выборки на выполнение очередной команды микропроцессор должен обязательно посмотреть содержимое сегментного регистра cs и только его. А в этом регистре, как мы знаем, содержится (пока еще не сдвинутый) физический адрес начала сегмента команд. Для получения адреса конкретной команды микропроцессору остается умножить содержимое cs на 16 (что означает сдвиг на четыре разряда) и сложить полученное 20-битное значение с 16-битным содержимым регистра ip. Примерно то же самое происходит и тогда, когда микропроцессор обрабатывает операнды в машинной команде. Если он видит, что операнд — это адрес (эффективный адрес, который является только частью физического адреса), то он знает, в каком сегменте его искать, — по умолчанию это сегмент, адрес начала которого записан в сегментном регистре ds.

А что же с сегментом стека? В контексте нашего рассмотрения нас интересуют *регистры sp и bp*. Если микропроцессор видит в качестве операнда (или его части, если операнд — выражение) один из этих регистров, то по умолчанию он формирует физический адрес операнда, используя в качестве его сегментной составляющей содержимое регистра ss. Это набор микропрограмм в блоке микропрограммного управления, каждая из которых выполняет одну из команд в системе машинных команд микропроцессора. Каждая микропрограмма работает по своему алгоритму. Изменить его, конечно, нельзя, но можно чуть-чуть подкорректировать. Делается это с помощью необязательного поля префикса машинной команды. Если мы согласны с тем, как работает команда, то это поле отсутствует. Если же мы хотим внести поправку (если, конечно, она допустима для конкретной команды) в алгоритм работы команды, то необходимо сформировать соответствующий префикс.

Префикс представляет собой однобайтовую величину, численное значение которой определяет ее назначение. Микропроцессор распознает по указанному значению, что этот байт является префиксом, и дальнейшая работа микропрограммы выполняется с учетом поступившего указания на корректировку ее работы. Сейчас нас интересует один из них — префикс замены (переопре-

деления) сегмента. Его назначение состоит в том, чтобы указать микропроцессору (а по сути, микропрограмме) на то, что мы не хотим использовать сегмент по умолчанию. Возможности для подобного переопределения, конечно, ограничены. Сегмент команд переопределить нельзя, адрес очередной исполняемой команды однозначно определяется парой cs:ip. А вот сегменты стека и данных — можно. Для этого и предназначен оператор «:». Транслятор ассемблера, обрабатывая этот оператор, формирует соответствующий однобайтовый префикс замены сегмента.

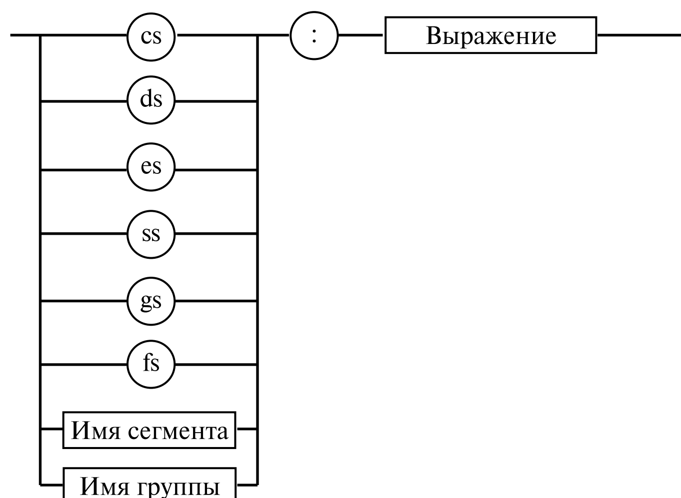


Рис. 14. Синтаксис оператора переопределения сегмента

8. Оператор именования типа структуры « . » (точка) также заставляет транслятор производить определенные вычисления, если он встречается в выражении.

9. Оператор получения сегментной составляющей адреса выражения seg возвращает физический адрес сегмента для выражения (рис. 15), в качестве которого могут выступать метка, переменная, имя сегмента, имя группы или некоторое символическое имя.

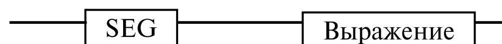


Рис. 15. Синтаксис оператора получения сегментной составляющей

10. Оператор получения смещения выражения offset позволяет получить значение смещения выражения (рис. 16) в байтах относительно начала того сегмента, в котором выражение определено.

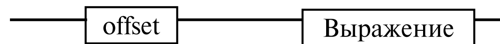


Рис. 16. Синтаксис оператора получения смещения

Как и в языках высокого уровня, выполнение операторов ассемблера при вычислении выражений осуществляется в соответствии с их приоритетами (табл. 4). Операции с одинаковыми приоритетами выполняются последовательно слева направо. Изменение порядка выполнения возможно путем расстановки круглых скобок, которые имеют наивысший приоритет.

Таблица 4

Операторы и их приоритет

Оператор	**Приоритет**
	1
.	2
:	3
ptr, offset, seg, type, this	4
high, low	5
+, – (унарные)	6
*, /, mod, shl, shr	7
+, –, (бинарные)	8
eq, ne, lt, le, gt, ge	9
not	10
and	11
or, xor	12
short, type	13

3. Директивы сегментации

В ходе предыдущего обсуждения мы выяснили все основные правила записи команд и операндов в программе на ассемблере. Открытым остался вопрос о том, как правильно оформить после-

довательность команд, чтобы транслятор мог их обработать, а микропроцессор — выполнить.

При рассмотрении архитектуры микропроцессора мы узнали, что он имеет шесть сегментных регистров, посредством которых может одновременно работать:

1) с одним сегментом кода;
2) с одним сегментом стека;
3) с одним сегментом данных;
4) с тремя дополнительными сегментами данных.

Еще раз вспомним, что физически сегмент представляет собой область памяти, занятую командами и (или) данными, адреса которых вычисляются относительно значения в соответствующем сегментном регистре.

Синтаксическое описание сегмента на ассемблере представляет собой конструкцию, изображенную на рисунке 17:

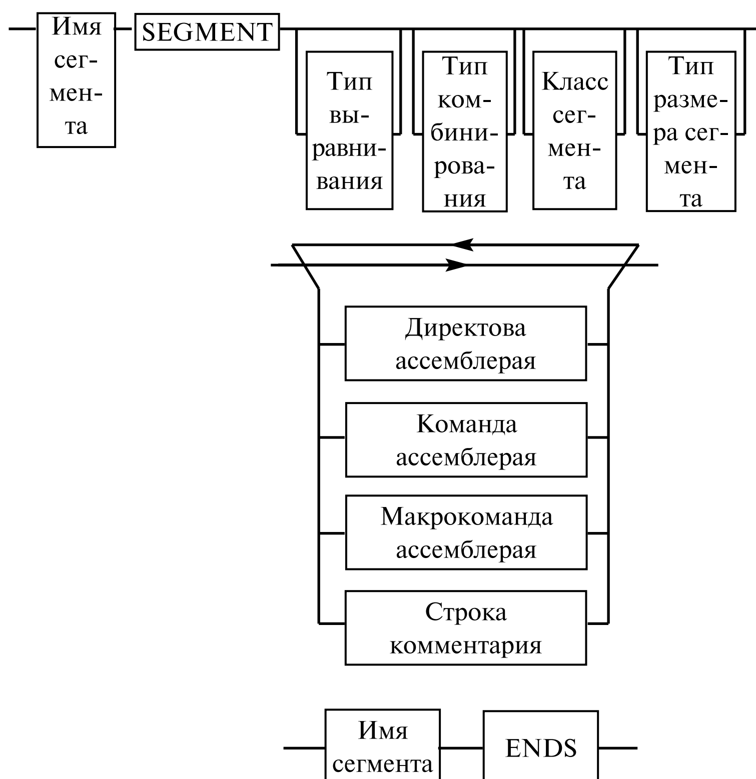


Рис. 17. Синтаксис описания сегмента

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

Рассмотрим их подробнее.

1. Атрибут выравнивания сегмента (тип выравнивания) сообщает компоновщику о том, что нужно обеспечить размещение начала сегмента на заданной границе. Это важно, поскольку при правильном выравнивании доступ к данным в процессорах i80x86 выполняется быстрее. Допустимые значения этого атрибута следующие:

1) BYTE — выравнивание не выполняется. Сегмент может начинаться с любого адреса памяти;

2) WORD — сегмент начинается по адресу, кратному двум, т. е. последний (младший) значащий бит физического адреса равен 0 (выравнивание на границу слова);

3) DWORD — сегмент начинается по адресу, кратному четырем, т. е. два последних (младших) значащих бита равны 0 (выравнивание на границу двойного слова);

4) PARA — сегмент начинается по адресу, кратному 16, т. е. последняя шестнадцатеричная цифра адреса должна быть 0h (выравнивание на границу параграфа);

5) PAGE — сегмент начинается по адресу, кратному 256, т. е. две последние шестнадцатеричные цифры должны быть 00h (выравнивание на границу 256-байтной страницы);

6) MEMPAGE — сегмент начинается по адресу, кратному 4 Кбайт, т. е. три последние шестнадцатеричные цифры должны быть 000h (адрес следующей 4-Кбайтной страницы памяти).

По умолчанию тип выравнивания имеет значение PARA .

2. Атрибут комбинирования сегментов (комбинаторный тип) сообщает компоновщику, как нужно комбинировать сегменты различных модулей, имеющие одно и то же имя. Значениями атрибута комбинирования сегмента могут быть:

1) PRIVATE — сегмент не будет объединяться с другими сегментами с тем же именем вне данного модуля;

2) PUBLIC — заставляет компоновщик соединить все сегменты с одинаковыми именами. Новый объединенный сегмент будет целым и непрерывным. Все адреса (смещения) объектов, а это могут быть в зависимости от типа сегмента команды и данные, будут вычисляться относительно начала этого нового сегмента;
3) COMMON — располагает все сегменты с одним и тем же именем по одному адресу. Все сегменты с данным именем будут перекрываться и совместно использовать память. Размер полученного в результате сегмента будет равен размеру самого большого сегмента;
4) AT xxxx — располагает сегмент по абсолютному адресу параграфа (параграф — объем памяти, кратный 16; поэтому последняя шестнадцатеричная цифра адреса параграфа равна 0). Абсолютный адрес параграфа задается выражением xxx. Компоновщик располагает сегмент по заданному адресу памяти (это можно использовать, например, для доступа к видеопамяти или области ПЗУ), учитывая атрибут комбинирования. Физически это означает, что сегмент при загрузке в память будет расположен, начиная с этого абсолютного адреса параграфа, но для доступа к нему в соответствующий сегментный регистр должно быть загружено заданное в атрибуте значение. Все метки и адреса в определенном таким образом сегменте отсчитываются относительно заданного абсолютного адреса;
5) STACK — определение сегмента стека. Заставляет компоновщик соединить все одноименные сегменты и вычислять адреса в этих сегментах относительно регистра ss. Комбинированный тип STACK (стек) аналогичен комбинированному типу PUBLIC, за исключением того, что регистр ss является стандартным сегментным регистром для сегментов стека. Регистр sp устанавливается на конец объединенного сегмента стека. Если не указано ни одного сегмента стека, компоновщик выдаст предупреждение, что стековый сегмент не найден. Если сегмент стека создан, а комбинированный тип STACK не используется, программист должен явно загрузить в регистр ss адрес сегмента (подобно тому, как это делается для регистра ds).

По умолчанию атрибут комбинирования принимает значение PRIVATE.

3. Атрибут класса сегмента (тип класса) — это заключенная в кавычки строка, помогающая компоновщику определить соответ-

ствующий порядок следования сегментов при собирании программы из сегментов нескольких модулей. Компоновщик объединяет вместе в памяти все сегменты с одним и тем же именем класса (имя класса в общем случае может быть любым, но лучше, если оно будет отражать функциональное назначение сегмента). Типичным примером использования имени класса является объединение в группу всех сегментов кода программы (обычно для этого используется *класс «code»*). С помощью механизма типизации класса можно группировать также сегменты инициализированных и неинициализированных данных.

4. Атрибут размера сегмента. Для процессоров i80386 и выше сегменты могут быть 16- или 32-разрядными. Это влияет прежде всего на размер сегмента и порядок формирования физического адреса внутри него. Атрибут может принимать следующие значения:

1) USE16 — это означает, что сегмент допускает 16-разрядную адресацию. При формировании физического адреса может использоваться только 16-разрядное смещение. Соответственно, такой сегмент может содержать до 64 Кбайт кода или данных;
2)USE32 — сегмент будет 32-разрядным. При формировании физического адреса может использоваться 32-разрядное смещение. Поэтому такой сегмент может содержать до 4 Гбайт кода или данных.

Все сегменты сами по себе равноправны, так как директивы SEGMENT и ENDS не содержат информации о функциональном назначении сегментов. Для того чтобы использовать их как сегменты кода, данных или стека, необходимо предварительно сообщить транслятору об этом, для чего используют специальную директиву ASSUME, имеющую формат, показанный на рис. 18. Эта директива сообщает транслятору о том, какой сегмент к какому сегментному регистру привязан. В свою очередь, это позволит транслятору корректно связывать символические имена, определенные в сегментах. Привязка сегментов к сегментным регистрам осуществляется с помощью операндов этой директивы, в которых имя_сегмента должно быть именем сегмента, определенным в исходном тексте программы директивой SEGMENT или ключевым *словом nothing*. Если в качестве операнда используется только ключевое *слово nothing*, то предшествующие назначения сегментных регистров анну-

лируются, причем сразу для всех шести сегментных регистров. Но ключевое слово nothing можно использовать вместо аргумента имя сегмента; в этом случае будет выборочно разрываться связь между сегментом с именем имя сегмента и соответствующим сегментным регистром (см. рис. 18).

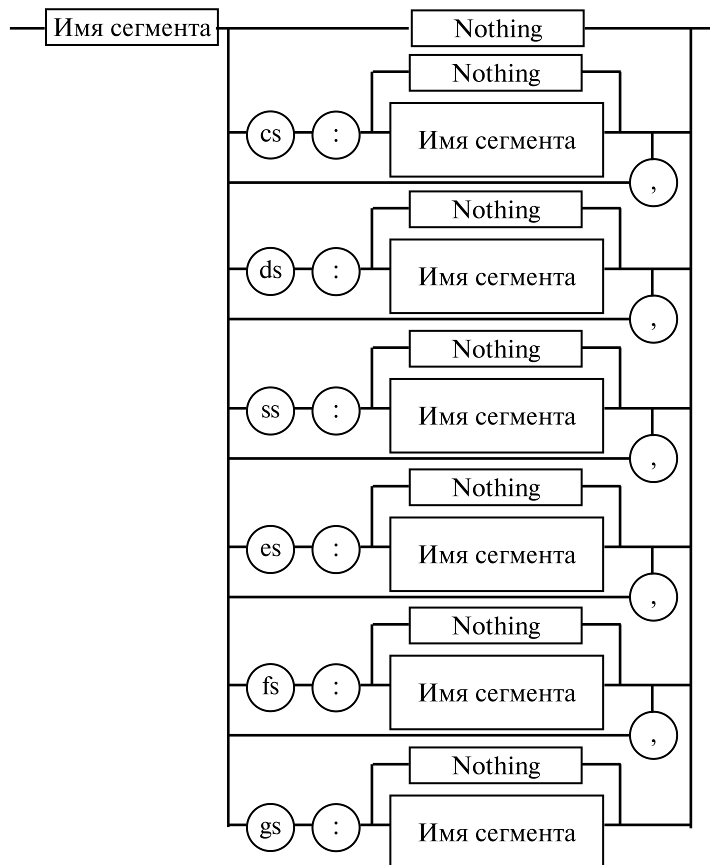


Рис. 18. Директива ASSUME

Для простых программ, содержащих по одному сегменту для кода, данных и стека, хотелось бы упростить ее описание. Для этого в трансляторы MASM и TASM ввели возможность использования упрощенных директив сегментации. Но здесь возникла проблема, связанная с тем, что необходимо было как-то компенсировать невозможность напрямую управлять размещением и комбинирова-

нием сегментов. Для этого совместно с упрощенными директивами сегментации стали использовать директиву указания модели памяти MODEL, которая частично стала управлять размещением сегментов и выполнять функции директивы ASSUME (поэтому при использовании упрощенных директив сегментации директиву ASSUME можно не использовать). Эта директива связывает сегменты, которые в случае использования упрощенных директив сегментации имеют предопределенные имена, с сегментными регистрами (хотя явно инициализировать ds все равно придется).

Синтаксис директивы MODEL показан на рисунке 19.

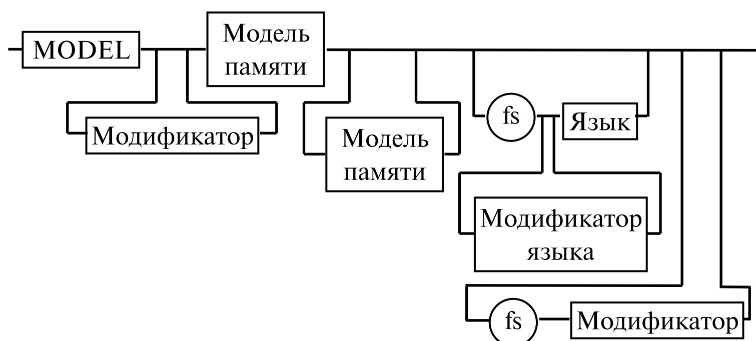


Рис. 19. Синтаксис директивы MODEL

Обязательным параметром директивы MODEL является модель памяти. Этот параметр определяет модель сегментации памяти для программного модуля. Предполагается, что программный модуль может иметь только определенные типы сегментов, которые определяются упомянутыми нами ранее упрощенными директивами описания сегментов. Эти директивы приведены в таблице 5.

Упрощенные директивы определения сегмента *Таблица 5*

Формат директивы (режим MASM)	**Формат директивы (режим IDEAL)**	**Назначение**
.CODE [имя]	CODESEG[имя]	Начало или продолжение сегмента кода

Окончание таб. 5

Формат директивы (режим MASM)	**Формат директивы (режим IDEAL)**	**Назначение**
.DATA	DATASEG	Начало или продолжение сегмента инициализированных данных. Также используется для определения данных типа near
.CONST	CONST	Начало или продолжение сегмента постоянных данных (констант) модуля
.DATA?	UDATASEG	Начало или продолжение сегмента неинициализированных данных. Также используется для определения данных типа near
.STACK [размер]	STACK [размер]	Начало или продолжение сегмента стека модуля. Параметр [размер] задает размер стека
.FARDATA [имя]	FARDATA [имя]	Начало или продолжение сегмента инициализированных данных типа far
.FARDATA? [имя]	UFARDATA [имя]	Начало или продолжение сегмента неинициализированных данных типа far

Наличие в некоторых директивах параметра [имя] говорит о том, что возможно определение нескольких сегментов этого типа. С другой стороны, наличие нескольких видов сегментов данных обусловлено требованием обеспечить совместимость с некоторыми компиляторами языков высокого уровня, которые создают разные сегменты данных для инициализированных и неинициализированных данных, а также констант.

При использовании директивы MODEL транслятор делает доступными несколько идентификаторов, к которым можно обращаться во время работы программы, с тем, чтобы получить информацию о тех или иных характеристиках данной модели памяти (табл. 7). Перечислим эти идентификаторы и их значения (табл. 6).

Идентификаторы, создаваемые директивой MODEL *Таблица 6.*

Имя идентификатора	Значение переменной
@code	Физический адрес сегмента кода
@data	Физический адрес сегмента данных типа near
@fardata	Физический адрес сегмента данных типа far
@fardata?	Физический адрес сегмента неинициализированных данных типа far
@curseg	Физический адрес сегмента неинициализированных данных типа far
@stack	Физический адрес сегмента стека

Теперь можно закончить обсуждение директивы MODEL. Операнды директивы MODEL используют для задания модели памяти, которая определяет набор сегментов программы, размеры сегментов данных и кода, способ связывания сегментов и сегментных регистров. В таблице 7 приведены некоторые значения параметра «модель памяти» директивы MODEL.

Модели памяти *Таблица 7*

Модель	Тип кода	Тип данных	Назначение модели
TINY	near	near	Код и данные объединены в одну группу с именем DGROUP. Используется для создания программ формата .com.
SMALL	near	near	Код занимает один сегмент, данные объединены в одну группу с именем DGROUP. Эту модель обычно используют для большинства программ на ассемблере
MEDIUM	far	near	Код занимает несколько сегментов, по одному на каждый объединяемый программный модуль. Все ссылки на передачу управления — типа far. Данные объединены в одной группе; все ссылки на них — типа near
COMPACT	near	far	Код в одном сегменте; ссылка на данные — типа far

Окончание табл. 7

Модель	Тип кода	Тип данных	Назначение модели
LARGE	far	far	Код в нескольких сегментах, по одному на каждый объединяемый программный модуль

Параметр «модификатор» директивы MODEL позволяет уточнить некоторые особенности использования выбранной модели памяти (табл. 8).

Модификаторы модели памяти *Таблица 8*

Значение модификатора	Назначение
use16	Сегменты выбранной модели используются как 16-битные (если соответствующей директивой указан процессор i80386 или i80486)
use32	Сегменты выбранной модели используются как 32-битные (если соответствующей директивой указан процессор i80386 или i80486)
dos	Программа будет работать в MS-DOS

Необязательные параметры «язык» и «модификатор языка» определяют некоторые особенности вызова процедур. Необ-ходимость в использовании этих параметров появляется при написании и связывании программ на различных языках программирования.

Описанные нами стандартные и упрощенные директивы сегментации не исключают друг друга. Стандартные директивы используются, когда программист желает получить полный контроль над размещением сегментов в памяти и их комбинированием с сегментами других модулей.

Упрощенные директивы целесообразно использовать для простых программ и программ, предназначенных для связывания с программными модулями, написанными на языках высокого уровня. Это позволяет компоновщику эффективно связывать модули разных языков за счет стандартизации связей и управления.

ЛЕКЦИЯ № 17. Структуры команд на Ассемблере

1. Структура машинной команды

Машинная команда представляет собой закодированное по определенным правилам указание микропроцессору на выполнение некоторой операции или действия. Каждая команда содержит элементы, определяющие:

1) что делать? (Ответ на этот вопрос дает элемент команды, называемый кодом операции (КОП).);

2) объекты, над которыми нужно что-то делать (эти элементы называются операндами);

3) как делать? (Эти элементы называются типами операндов — обычно задаются неявно).

Приведенный на рисунке 20 формат машинной команды является самым общим. Максимальная длина машинной команды — 15 байт. Реальная команда может содержать гораздо меньшее количество полей, вплоть до одного — только КОП.

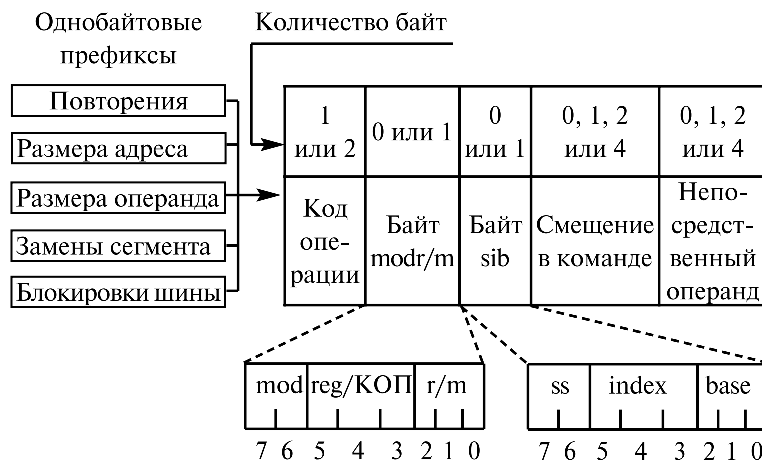


Рис. 20. Формат машинной команды

Опишем назначения полей машинной команды.

1. Префиксы.

Необязательные элементы машинной команды, каждый из которых состоит из 1 байта или может отсутствовать. В памяти префиксы предшествуют команде. Назначение префиксов — модифицировать операцию, выполняемую командой. Прикладная программа может использовать следующие типы префиксов:

1) префикс замены сегмента. В явной форме указывает, какой сегментный регистр используется в данной команде для адресации стека или данных. Префикс отменяет выбор сегментного регистра по умолчанию. Префиксы замены сегмента имеют следующие значения:

а) 2eh — замена сегмента cs;

б) 36h — замена сегмента ss;

в) 3eh — замена сегмента ds;

г) 26h — замена сегмента es;

д) 64h — замена сегмента fs;

е) 65h — замена сегмента gs;

2) префикс разрядности адреса уточняет разрядность адреса (32- или 16-разрядный). Каждой команде, в которой используется адресный операнд, ставится в соответствие разрядность адреса этого операнда. Этот адрес может иметь разрядность 16 или 32 бит. Если разрядность адреса для данной команды 16 бит, это означает, что команда содержит 16-разрядное смещение (рис. 20), оно соответствует 16-разрядному смещению адресного операнда относительно начала некоторого сегмента. В контексте рисунка 21 это смещение называется эффективный адрес. Если разрядность адреса 32 бит, это означает, что команда содержит 32-разрядное смещение (рис. 20), оно соответствует 32-разрядному смещению адресного операнда относительно начала сегмента, и по его значению формируется 32-битное смещение в сегменте. С помощью префикса разрядности адреса можно изменить действующее по умолчанию значение разрядности адреса. Это изменение будет касаться только той команды, которой предшествует префикс;

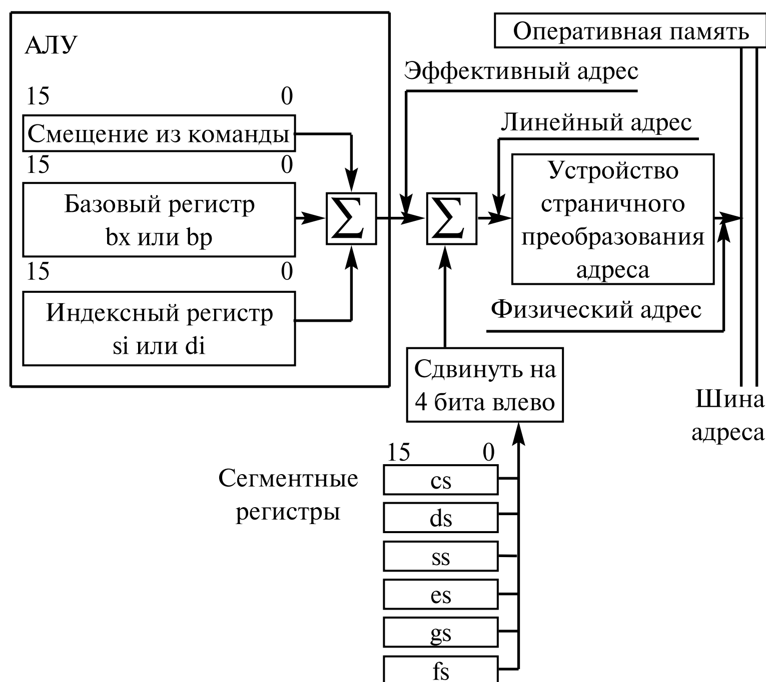


Рис. 21. Механизм формирования физического адреса в реальном режиме

3) префикс разрядности операнда аналогичен префиксу разрядности адреса, но указывает на разрядность операндов (32- или 16-разрядные), с которыми работает команда. В соответствии с какими правилами устанавливаются значения атрибутов разрядности адреса и операндов по умолчанию?

В реальном режиме и режиме виртуального i8086 значения этих атрибутов — 16 бит. В защищенном режиме значения атрибутов зависят от состояния бита D в дескрипторах исполняемых сегментов. Если D = 0, то значения атрибутов, действующие по умолчанию, равны 16 бит; если D = 1, то 32 бит.

Значения префиксов разрядности операнда 66h и разрядности адреса 67h. С помощью префикса разрядности адреса в реальном режиме можно использовать 32-разрядную адресацию, но при этом необходимо помнить об ограниченности размера сегмента величиной 64 Кбайт. Аналогично префиксу разрядности адреса вы можете использовать префикс разрядности операнда в реальном режиме для работы с 32-разрядными операндами (к примеру, в арифметических командах);

4) префикс повторения используется с цепочечными командами (командами обработки строк). Этот префикс «зацикливает» команду для обработки всех элементов цепочки. Система команд поддерживает два типа префиксов:

а) безусловные (rep — 0f3h), заставляющие повторяться цепочечную команду некоторое количество раз;

б) условные (repe/repz — 0f3h, repne/repnz — 0f2h), которые при зацикливании проверяют некоторые флаги, и в результате проверки возможен досрочный выход из цикла.

2. Код операции.

Обязательный элемент, описывающий операцию, выполняемую командой. Многим командам соответствует несколько кодов операций, каждый из которых определяет нюансы выполнения операции. Последующие поля машинной команды определяют местоположение операндов, участвующих в операции, и особенности их использования. Рассмотрение этих полей связано со способами задания операндов в машинной команде и потому будет выполнено позже.

3. Байт режима адресации modr/m.

Значения этого байта определяет используемую форму адреса операндов. Операнды могут находиться в памяти в одном или двух регистрах. Если операнд находится в памяти, то байт modr/m определяет компоненты (смещение, базовый и индексный регистры), используемые для вычисления его эффективного адреса (рисунок 21). В защищенном режиме для определения местоположения операнда в памяти может дополнительно использоваться байт sib (Scale-Index-Base — масштаб-индекс-база). Байт modr/m состоит из трех полей (рис. 20):

1) поле mod определяет количество байт, занимаемых в команде адресом операнда (рис. 20 , поле смещение в команде).

Поле mod используется совместно с полем r/m , которое указывает способ модификации адреса операнда «смещение в команде». К примеру, если mod = 00, это означает, что поле смещение в команде отсутствует, и адрес операнда определяется содержимым базового и (или) индексного регистра. Какие именно регистры будут использоваться для вычисления эффективного адреса, определяется значением этого байта. Если mod = 01, это означает, что поле смещение в команде присутствует, занимает 1 байт и модифицируется содержимым базового и (или) индексного регистра. Если mod = 10, это означает, что поле смещение в ко-

манде присутствует, занимает 2 или 4 байта (в зависимости от действующего по умолчанию или определяемого префиксом размера адреса) и модифицируется содержимым базового и (или) индексного регистра. Если mod = 11, это означает, что операндов в памяти нет: они находятся в регистрах. Это же значение байта mod используется в случае, когда в команде применяется непосредственный операнд;

2) поле reg/коп определяет либо регистр, находящийся в команде на месте первого операнда, либо возможное расширение кода операции;

3) поле r/m используется совместно с полем mod и определяет либо регистр, находящийся в команде на месте первого операнда (если mod = 11), либо используемые для вычисления эффективного адреса (совместно с полем смещение в команде) базовые и индексные регистры.

4. Байт масштаб — индекс — база (байт sib).

Используется для расширения возможностей адресации операндов. На наличие байта sib в машинной команде указывает сочетание одного из значений 01 или 10 поля mod и значения поля r/m = 100. Байт sib состоит из трех полей:

1) поля масштаба ss. В этом поле размещается масштабный множитель для индексного компонента index, занимающего следующие 3 бита байта sib. В поле ss может содержаться одно из следующих значений: 1, 2, 4, 8.

При вычислении эффективного адреса на это значение будет умножаться содержимое индексного регистра;

2) поля index. Используется для хранения номера индексного регистра, который применяется для вычисления эффективного адреса операнда;

3) поля base. Используется для хранения номера базового регистра, который также применяется для вычисления эффективного адреса операнда. В качестве базового и индексного регистров могут использоваться практически все регистры общего назначения.

5. Поле смещения в команде.

8-, 16- или 32-разрядное целое число со знаком, представляющее собой, полностью или частично (с учетом вышеприведенных рассуждений), значение эффективного адреса операнда.

6. Поле непосредственного операнда.

Необязательное поле, представляющее собой 8-, 16- или 32-разрядный непосредственный операнд. Наличие этого поля, конечно, отражается на значении байта modr/m .

2. Способы задания операндов команды

Операнд задается неявно на микропрограммном уровне

В этом случае команда явно не содержит операндов. Алгоритм выполнения команды использует некоторые объекты по умолчанию (регистры, флаги в eflags и т. д.).

Например, команды cli и sti неявно работают с флагом прерывания if в регистре eflags, а команда xlat неявно обращается к регистру al и строке в памяти по адресу, определяемому парой регистров ds : bx.

Операнд задается в самой команде (непосредственный операнд)

Операнд находится в коде команды, т. е. является ее частью. Для хранения такого операнда в команде выделяется поле длиной до 32 бит (рисунок 20). Непосредственный операнд может быть только вторым операндом (источником). Операнд-получатель может находиться либо в памяти, либо в регистре.

Например: mov ax,0ffffh пересылает в регистр ax шестнадцатеричную константу ffff. Команда add sum,2 складывает содержимое поля по адресу sum с целым числом 2 и записывает результат по месту первого операнда, т. е. в память.

Операнд находится в одном из регистров

Регистровые операнды указываются именами регистров. В качестве регистров могут использоваться:

1) 32-разрядные регистры EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP;
2) 16-разрядные регистры AX, BX, CX, DX, SI, DI, SP, BP;
3) 8-разрядные регистры AH, AL, BH, BL, CH, CL, DH, DL;
4) сегментные регистры CS, DS, SS, ES, FS, GS.

Например, команда add ax,bx складывает содержимое регистров ax и bx и записывает результат в bx. Команда dec si уменьшает содержимое si на 1.

Операнд располагается в памяти

Это наиболее сложный и в то же время наиболее гибкий способ задания операндов. Он позволяет реализовать следующие два основных вида адресации: прямую и косвенную.

В свою очередь, косвенная адресация имеет следующие разновидности:

1) косвенную базовую адресацию; другое ее название — регистровая косвенная адресация;
2) косвенную базовую адресацию со смещением;
3) косвенную индексную адресацию со смещением;
4) косвенную базовую индексную адресацию;
5) косвенную базовую индексную адресацию со смещением.

Операндом является порт ввода/вывода

Помимо адресного пространства оперативной памяти, микропроцессор поддерживает адресное пространство ввода-вывода, которое используется для доступа к устройствам ввода-вывода. Объем адресного пространства ввода-вывода составляет 64 Кбайт. Для любого устройства компьютера в этом пространстве выделяются адреса. Конкретное значение адреса в пределах этого пространства называется портом ввода-вывода. Физически порту ввода-вывода соответствует аппаратный регистр (не путать с регистром микропроцессора), доступ к которому осуществляется с помощью специальных команд ассемблера in и out.

Например:

```
in al,60h; ввести байт из порта 60h
```

Регистры, адресуемые с помощью порта ввода-вывода, могут иметь разрядность 8, 16 или 32 бит, но для конкретного порта разрядность регистра фиксирована. Команды in и out работают с фиксированной номенклатурой объектов. В качестве источника информации или получателя применяются так называемые регистрыаккумуляторы EAX, AX, AL. Выбор регистра определяется разрядностью порта. Номер порта может задаваться непосредст-венным операндом в командах in и out или значением в регистре DX. Последний способ позволяет динамически определить номер порта в программе.

Операнд находится в стеке

Команды могут совсем не иметь операндов, иметь один или два операнда. Большинство команд требуют двух операндов, один из которых является операндом-источником, а второй — операндом назначения. Важно то, что один операнд может располагаться в регистре или памяти, а второй операнд обязательно должен находиться в регистре или непосредственно в команде. Непосредственный операнд может быть только операндом-источником. В двухоперандной машинной команде возможны следующие сочетания операндов:

1) регистр — регистр;
2) регистр — память;
3) память — регистр;
4) непосредственный операнд — регистр;
5) непосредственный операнд — память.

У данного правила есть исключения, которые касаются:

1) команд работы с цепочками, которые могут перемещать данные из памяти в память;
2) команд работы со стеком, которые могут переносить данные из памяти в стек, также находящийся в памяти;
3) команд типа умножения, которые, кроме операнда, указанного в команде, используют еще и второй, неявный операнд.

Из перечисленных сочетаний операндов наиболее часто употребляются регистр — память и память — регистр. Ввиду их важности рассмотрим их подробнее. Обсуждение мы будем сопровождать примерами команд ассемблера, которые будут показывать, как изменяется формат команды ассемблера при применении того или иного вида адресации. В связи с этим посмотрите еще раз на рисеуjr 21, на котором показан принцип формирования физического адреса на адресной шине микропроцессора. Видно, что адрес операнда формируется как сумма двух составляющих — сдвинутого на 4 бита содержимого сегментного регистра и 16-битного эффективного адреса, который в общем случае вычисляется как сумма трех компонентов: базы, смещения и индекса.

3. Способы адресации

Перечислим и затем рассмотрим особенности основных видов адресации операндов в памяти:

1) прямую адресацию;

2) косвенную базовую (регистровую) адресацию;
3) косвенную базовую (регистровую) адресацию со смещением;
4) косвенную индексную адресацию со смещением;
5) косвенную базовую индексную адресацию;
6) косвенную базовую индексную адресацию со смещением.

Прямая адресация

Это простейший вид адресации операнда в памяти, так как эффективный адрес содержится в самой команде и для его формирования не используется никаких дополнительных источников или регистров. Эффективный адрес берется непосредственно из поля смещения машинной команды (см. рис. 20), которое может иметь размер 8, 16, 32 бит. Это значение однозначно определяет байт, слово или двойное слово, расположенные в сегменте данных.

Прямая адресация может быть двух типов.

Относительная прямая адресация

Используется для команд условных переходов, для указания относительного адреса перехода. Относительность такого перехода заключается в том, что в поле смещения машинной команды содержится 8-, 16- или 32-битное значение, которое в результате работы команды будет складываться с содержимым регистра указателя команд ip/eip. В результате такого сложения получается адрес, по которому и осуществляется переход.

Абсолютная прямая адресация

В этом случае эффективный адрес является частью машинной команды, но формируется этот адрес только из значения поля смещения в команде. Для формирования физического адреса операнда в памяти микропроцессор складывает это поле со сдвинутым на 4 бита значением сегментного регистра. В команде ассемблера можно использовать несколько форм такой адресации.

Но такая адресация применяется редко — обычно используемым ячейкам в программе присваиваются символические имена. В процессе трансляции ассемблер вычисляет и подставляет значения смещений этих имен в формируемую им машинную команду в поле «смещение в команде». В итоге получается так, что машин-

ная команда прямо адресует свой операнд, имея, фактически, в одном из своих полей значение эффективного адреса.

Остальные виды адресации относятся к косвенным. Слово «косвенный» в названии этих видов адресации означает то, что в самой команде может находиться лишь часть эффективного адреса, а остальные его компоненты находятся в регистрах, на которые указывают своим содержимым байт modr/m и, возможно, байт sib.

Косвенная базовая (регистровая) адресация

При такой адресации эффективный адрес операнда может находиться в любом из регистров общего назначения, кроме sp/esp и bp/ebp (это специфические регистры для работы с сегментом стека). Синтаксически в команде этот режим адресации выражается заключением имени регистра в квадратные скобки [ ]. К примеру, команда mov ax,[ecx] помещает в регистр ax содержимое слова по адресу из сегмента данных со смещением, хранящимся в регистре ecx. Так как содержимое регистра легко изменить в ходе работы программы, данный способ адресации позволяет динамически назначить адрес операнда для некоторой машинной команды. Это свойство очень полезно, например, для организации циклических вычислений и для работы с различными структурами данных типа таблиц или массивов.

Косвенная базовая (регистровая) адресация со смещением

Этот вид адресации является дополнением предыдущего и предназначен для доступа к данным с известным смещением относительно некоторого базового адреса. Этот вид адресации удобно использовать для доступа к элементам структур данных, когда смещение элементов известно заранее, на стадии разработки программы, а базовый (начальный) адрес структуры должен вычисляться динамически, на стадии выполнения программы. Модификация содержимого базового регистра позволяет обратиться к одноименным элементам различных экземпляров однотипных структур данных.

К примеру, команда mov ax,[edx+3h] пересылает в регистр ax слова из области памяти по адресу: содержимое edx + 3h.

Команда mov ax,mas[dx] пересылает в регистр ax слово по адресу: содержимое dx плюс значение идентификатора mas (не забывайте, что транслятор присваивает каждому идентификатору значение, равное смещению этого идентификатора относительно начала сегмента данных).

Косвенная индексная адресация со смещением

Этот вид адресации очень похож на косвенную базовую адресацию со смещением. Здесь также для формирования эффективного адреса используется один из регистров общего назначения. Но индексная адресация обладает одной интересной особенностью, которая очень удобна для работы с массивами. Она связана с возможностью так называемого масштабирования содержимого индексного регистра. Что это такое?

Посмотрите на рисунок 20. Нас интересует байт sib. При обсуждении структуры этого байта мы отмечали, что он состоит из трех полей. Одно из этих полей — поле масштаба ss, на значение которого умножается содержимое индексного регистра.

К примеру, в команде mov ax,mas[si*2] значение эффективного адреса второго операнда вычисляется выражением mas+(si)*2. В связи с тем, что в ассемблере нет средств для организации индексации массивов, то программисту своими силами приходится ее организовывать.

Наличие возможности масштабирования существенно помогает в решении этой проблемы, но при условии, что размер элементов массива составляет 1, 2, 4 или 8 байт.

Косвенная базовая индексная адресация

При этом виде адресации эффективный адрес формируется как сумма содержимого двух регистров общего назначения: базового и индексного. В качестве этих регистров могут применяться любые регистры общего назначения, при этом часто используется масштабирование содержимого индексного регистра.

Косвенная базовая индексная адресация со смещением

Этот вид адресации является дополнением косвенной индексной адресации. Эффективный адрес формируется как сумма трех составляющих: содержимого базового регистра, содержимого индексного регистра и значения поля смещения в команде.

К примеру, команда mov eax,[esi+5][edx] пересылает в регистр eax двойное слово по адресу: (esi) + 5 + (edx).

Команда add ax,array[esi][ebx] производит сложение содержимого регистра ax с содержимым слова по адресу: значение идентификатора array + (esi) + (ebx).

ЛЕКЦИЯ № 18. Команды

1. Команды пересылки данных

Для удобства практического применения и отражения их специфики команды данной группы удобнее рассматривать в соответствии с их функциональным назначением, согласно которому их можно разбить на следующие группы команд:

1) пересылки данных общего назначения;
2) ввода-вывода в порт;
3) работы с адресами и указателями;
4) преобразования данных;
5) работы со стеком.

Команды пересылки данных общего назначения

К этой группе относятся следующие команды:

1) **mov** — это основная команда пересылки данных. Она реализует самые разнообразные варианты пересылки.

Отметим особенности применения этой команды:

а) командой *mov* нельзя осуществить пересылку из одной области памяти в другую. Если такая необходимость возникает, то нужно использовать в качестве промежуточного буфера любой доступный в данный момент регистр общего назначения;

б) нельзя загрузить в сегментный регистр значение непосредственно из памяти. Поэтому для выполнения такой загрузки нужно использовать промежуточный объект. Это может быть регистр общего назначения или стек;

в) нельзя переслать содержимое одного сегментного регистра в другой сегментный регистр. Это объясняется тем, что в системе команд нет соответствующего кода операции. Но необходимость в таком действии часто возникает. Выполнить такую пересылку можно, используя в качестве промежуточных все те же регистры общего назначения;

г) нельзя использовать сегментный регистр CS в качестве операнда назначения. Причина здесь простая. Дело в том, что в архитектуре микропроцессора пара cs:ip всегда содержит адрес команды, которая должна выполняться следующей. Изменение командой mov содержимого регистра CS фактически означало бы операцию перехода, а не пересылки, что недопустимо.

2) **xchg** — применяют для двунаправленной пересылки данных. Для этой операции можно, конечно, применить последовательность из нескольких команд *mov*, но из-за того, что операция обмена используется довольно часто, разработчики системы команд микропроцессора посчитали нужным ввести отдельную команду обмена xchg. Естественно, что операнды должны иметь один тип. Не допускается (как и для всех команд ассемблера) обменивать между собой содержимое двух ячеек памяти.

Команды ввода-вывода в порт

Посмотрите на рисунок 22. На нем показана сильно упрощенная, концептуальная схема управления оборудованием компьютера.

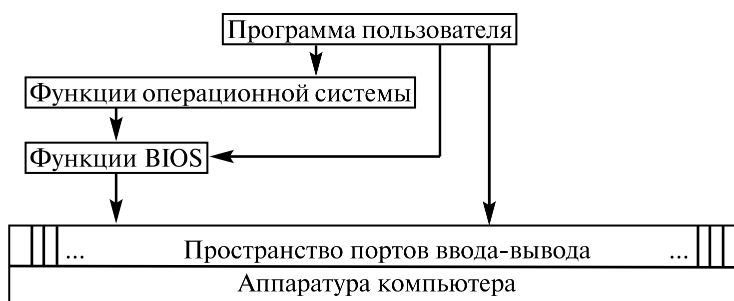


Рис. 22. Концептуальная схема управления оборудованием компьютера

Как видно из рисунка 22, самым нижним уровнем является уровень BIOS, на котором работа с оборудованием ведется напрямую через порты. Тем самым реализуется концепция независимости от оборудования. При замене оборудования необходимо будет лишь подправить соответствующие функции BIOS, переориентировав их на новые адреса и логику работы портов.

Принципиально управлять устройствами напрямую через порты несложно. Сведения о номерах портов, их разрядности, формате управляющей информации приводятся в техническом описании устройства. Необходимо знать лишь конечную цель своих действий, алгоритм, в соответствии с которым работает конкретное устройство, и порядок программирования его портов, т. е., фактически, нужно знать, что и в какой последовательности нужно послать в порт (при записи в него) или считать из него (при чтении) и как следует трактовать эту информацию. Для этого достаточно всего двух команд, присутствующих в системе команд микропроцессора:

1) **in аккумулятор, номер_порта** — ввод в аккумулятор из порта с номером номер_порта;
2) **out порт, аккумулятор** — вывод содержимого аккумулятора в порт с номером номер_порта.

Команды работы с адресами и указателями памяти

При написании программ на ассемблере производится интенсивная работа с адресами операндов, находящимися в памяти. Для поддержки такого рода операций есть специальная группа команд, в которую входят следующие команды:

1) **lea назначение, источник** — загрузка эффективного адреса;
2) **lds назначение, источник** — загрузка указателя в регистр сегмента данных ds;
3) **les назначение, источник** — загрузка указателя в регистр дополнительного сегмента данных es;
4) **lgs назначение, источник** — загрузка указателя в регистр дополнительного сегмента данных gs;
5) **lfs назначение, источник** — загрузка указателя в регистр дополнительного сегмента данных fs;
6) **lss назначение, источник** — загрузка указателя в регистр сегмента стека ss.

Команда *lea* похожа на команду *mov* тем, что она также производит пересылку. Однако команда *lea* производит пересылку не данных, а эффективного адреса данных (т. е. смещения данных относительно начала сегмента данных) в регистр, указанный операндом *назначение*.

Часто для выполнения некоторых действий в программе недостаточно знать значение одного лишь эффективного адреса дан-

ных, а необходимо иметь полный указатель на данные. Полный указатель на данные состоит из сегментной составляющей и смещения. Все остальные команды этой группы позволяют получить в паре регистров такой полный указатель на операнд в памяти. При этом *имя сегментного регистра*, в который помещается сегментная составляющая адреса, определяется кодом операции. Соответственно *смещение* помещается в регистр общего назначения, указанный операндом *назначение*.

Но не все так просто с операндом *источник*. На самом деле, в команде в качестве источника нельзя указывать непосредственно имя операнда в памяти, на который мы бы хотели получить указатель. Предварительно необходимо получить само значение полного указателя в некоторой области памяти и указать в команде получения полный адрес имени этой области. Для выполнения этого действия необходимо вспомнить директивы резервирования и инициализации памяти.

При применении этих директив возможен частный случай, когда в поле операндов указывается имя другой директивы определения данных (фактически, имя переменной). В этом случае в памяти формируется адрес этой переменной. Какой адрес будет сформирован (эффективный или полный), зависит от применяемой директивы. Если это dw, то в памяти формируется только 16-битное значение эффективного адреса, если же dd — в память записывается полный адрес. Размещение этого адреса в памяти следующее: в младшем слове находится смещение, в старшем — 16-битная сегментная составляющая адреса.

Например, при организации работы с цепочкой символов удобно поместить ее начальный адрес в некоторый регистр и далее в цикле модифицировать это значение для последовательного доступа к элементам цепочки.

Необходимость использования команд получения полного указателя данных в памяти, т. е. адреса сегмента и значения смещения внутри сегмента, возникает, в частности, при работе с цепочками.

Команды преобразования данных

К этой группе можно отнести множество команд микропроцессора, но большинство из них имеет те или иные особенности,

которые требуют отнести их к другим функциональным группам. Поэтому из всей совокупности команд микропроцессора непосредственно к командам преобразования данных можно отнести только одну команду: **xlat [адрес_таблицы_перекодировки]**

Это очень интересная и полезная команда. Ее действие заключается в том, что она замещает значение в регистре al другим байтом из таблицы в памяти, расположенной по адресу, указанному операндом *адрес_таблицы_перекодировки*.

Слово «таблица» весьма условно, по сути, это просто строка байт. Адрес байта в строке, которым будет производиться замещение содержимого регистра al, определяется суммой (bx) + (al), т. е. содержимое al исполняет роль индекса в байтовом массиве.

При работе с командой xlat обратите внимание на следующий тонкий момент. Несмотря на то что в команде указывается адрес строки байт, из которой должно быть извлечено новое значение, этот адрес должен быть предварительно загружен (например, с помощью команды *lea*) в регистр bx. Таким образом, операнд *адрес_таблицы_перекодировки* на самом деле не нужен (необязательность операнда показана заключением его в квадратные скобки). Что касается строки *байт* (таблицы перекодировки), то она представляет собой область памяти размером от 1 до 255 байт (диапазон числа без знака в 8-битном регистре).

Команды работы со стеком

Эта группа представляет собой набор специализированных команд, ориентированных на организацию гибкой и эффективной работы со стеком.

Стек — это область памяти, специально выделяемая для временного хранения данных программы. Важность стека определяется тем, что для него в структуре программы предусмотрен отдельный сегмент. На тот случай, если программист забыл описать сегмент стека в своей программе, компоновщик tlink выдаст предупреждающее сообщение.

Для работы со стеком предназначены три регистра:

1) **ss** — сегментный регистр стека;

2) **sp/esp** — регистр указателя стека;

3) **bp/ebp** — регистр указателя базы кадра стека.

Размер стека зависит от режима работы микропроцессора и ограничивается 64 Кбайтами (или 4 Гбайтами в защищенном режиме).

В каждый момент времени доступен только один стек, адрес сегмента которого содержится в регистре SS. Этот стек называется текущим. Для того чтобы обратиться к другому стеку («переключить стек»), необходимо загрузить в регистр ss другой адрес. Регистр SS автоматически используется процессором для выполнения всех команд, работающих со стеком.

Перечислим еще некоторые особенности работы со стеком:

1) запись и чтение данных в стеке осуществляется в соответствии с принципом LIFO;

2) по мере записи данных в стек последний растет в сторону младших адресов. Эта особенность заложена в алгоритм команд работы со стеком;

3) при использовании регистров esp/sp и ebp/bp для адресации памяти ассемблер автоматически считает, что содержащиеся в нем значения представляют собой смещения относительно сегментного регистра ss.

В общем случае стек организован так, как показано на рисунке 23.

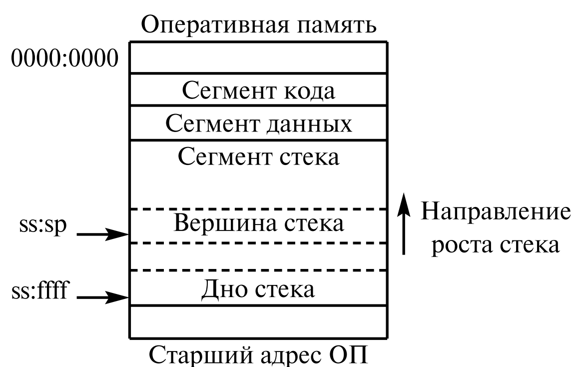


Рис. 23. Концептуальная схема организации стека

Для работы со стеком предназначены регистры SS, ESP/SP и EBP/BP. Эти регистры используются комплексно, и каждый из них имеет свое функциональное назначение.

Регистр ESP/SP всегда указывает на вершину стека, т. е. содержит смещение, по которому в стек был занесен последний элемент. Команды работы со стеком неявно изменяют этот регистр так, чтобы он указывал всегда на последний записанный в стек элемент. Если стек пуст, то значение esp равно адресу последнего байта сегмента, выделенного под стек. При занесении элемента в стек процессор уменьшает значение регистра esp, а затем записывает элемент по адресу новой вершины. При извлечении данных из стека процессор копирует элемент, расположенный по адресу вершины, а затем увеличивает значение регистра указателя стека esp. Таким образом, получается, что стек растет вниз, в сторону уменьшения адресов.

Что делать, если нам необходимо получить доступ к элементам не на вершине, а внутри стека? Для этого применяют регистр EBP. Регистр EBP — *регистр указателя базы кадра стека*.

Например, типичным приемом при входе в подпрограмму является передача нужных параметров путем записи их в стек. Если подпрограмма тоже активно работает со стеком, то доступ к этим параметрам становится проблематичным. Выход в том, чтобы после записи нужных данных в стек сохранить адрес вершины стека в указателе кадра (базы) стека — регистре EBP. Значение в EBP в дальнейшем можно использовать для доступа к переданным параметрам.

Начало стека расположено в старших адресах памяти. На рисунке 23 этот адрес обозначен парой ss:ffff. Смещение ffff приведено здесь условно. Реально это значение определяется величиной, которую программист задает при описании сегмента стека в своей программе.

Для организации работы со стеком существуют специальные команды записи и чтения.

1. **push источник** — запись значения *источник* в вершину стека.

Интерес представляет алгоритм работы этой команды, который включает следующие действия (рис. 24):

1) (sp) = (sp) — 2; значение sp уменьшается на 2;

2) значение из источника записывается по адресу, указываемому парой ss:sp.

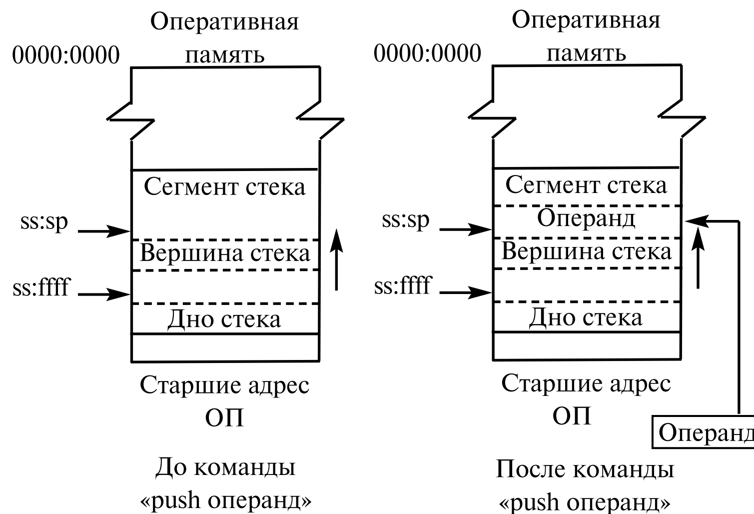


Рис. 24. Принцип работы команды push

2. **pop назначение** - запись значения из вершины стека по месту, указанному операндом *назначение*. Значение при этом «снимается» с вершины стека. Алгоритм работы команды pop обратен алгоритму команды push (рис. 25):

1) запись содержимого вершины стека по месту, указанному операндом *назначение*;
2) (sp) = (sp) + 2; увеличение значения sp.

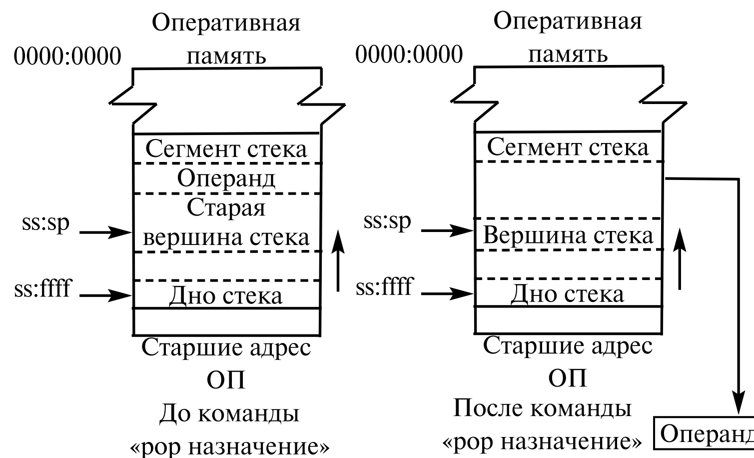


Рис. 25. Принцип работы команды pop

3. **pusha** — команда групповой записи в стек. По этой команде в стек последовательно записываются регистры ax, cx, dx, bx, sp, bp, si, di. Заметим, что записывается оригинальное содержимое sp, т. е. то, которое было до выдачи команды pusha (рис. 26).

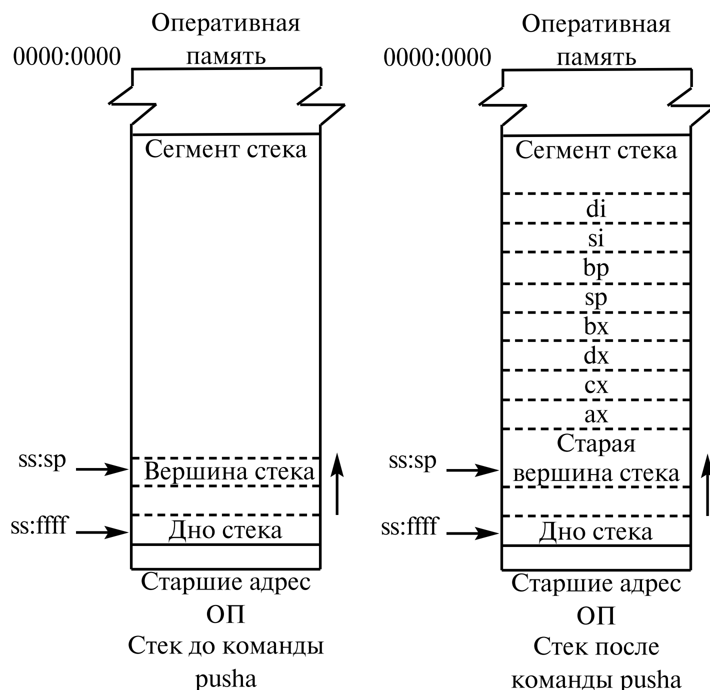


Рис. 26. Принцип работы команды pusha

4. **pushaw** — почти синоним команды pusha. В чем разница? Атрибут разрядности может принимать значение use16 или use32. Рассмотрим работу команд pusha и pushaw при каждом из этих атрибутов:

1) use16 — алгоритм работы pushaw аналогичен алгоритму pusha;

2) use32 — pushaw не изменяется (т. е. она нечувствительна к разрядности сегмента и всегда работает с регистрами размером в слово — ax, cx, dx, bx, sp, bp, si, di). Команда pusha чувствительна к установленной разрядности сегмента и при указании 32-разрядного сегмента работает с соответствующими 32-разрядными регистрами, т. е. eax, ecx, edx, ebx, esp, ebp, esi, edi.

5. **pushad** — выполняется аналогично команде pusha, но есть некоторые особенности.

Следующие три команды выполняют действия, обратные вышеописанным командам:

1) **popa;**
2) **popaw;**
3) **popad.**

Группа команд, описанная ниже, позволяет сохранить в стеке регистр флагов и записать слово или двойное слово в стеке. Отметим, что перечисленные ниже команды — единственные в системе команд микропроцессора, которые позволяют получить доступ (и которые нуждаются в этом доступе) ко всему содержимому регистра флагов.

1. **pushf** — сохраняет регистр флагов в стеке.

Работа этой команды зависит от атрибута размера сегмента:

1) use16 — в стек записывается регистр flags размером 2 байта;
2) use32 — в стек записывается регистр eflags размером 4 байта.

2. **pushfw** — сохранение в стеке регистра флагов размером в слово. Всегда работает как pushf с атрибутом use16.

3. **pushfd** — сохранение в стеке регистра флагов flags или eflags в зависимости от атрибута разрядности сегмента (т. е. то же, что и pushf).

Аналогично, следующие три команды выполняют действия, обратные рассмотренным выше операциям:

1) **popf;**
2) **popfw;**
3) **popfd.**

И в заключение отметим основные виды операции, когда использование стека практически неизбежно:

1) вызов подпрограмм;
2) временное сохранение значений регистров;
3) определение локальных переменных.

2. Арифметические команды

Микропроцессор может выполнять целочисленные операции и операции с плавающей точкой. Для этого в его архитектуре есть два отдельных блока:

1) устройство для выполнения целочисленных операций;
2) устройство для выполнения операций с плавающей точкой.

Каждое из этих устройств имеет свою систему команд. В принципе, целочисленное устройство может взять на себя многие функ-

ции устройства с плавающей точкой, но это потребует больших вычислительных затрат. Для большинства задач, использующих язык ассемблера, достаточно целочисленной арифметики.

Обзор группы арифметических команд и данных

Целочисленное вычислительное устройство поддерживает чуть больше десятка арифметических команд. На рисунке 27 приведена классификация команд этой группы.

Рис. 27. Классификация арифметических команд

Группа арифметических целочисленных команд работает с двумя типами чисел:

1) целыми двоичными числами. Числа могут иметь знаковый разряд или не иметь такового, т. е. быть числами со знаком или без знака;

2) целыми десятичными числами.

Рассмотрим машинные форматы, в которых хранятся эти типы данных.

Целые двоичные числа

Целое двоичное число с фиксированной точкой — это число, закодированное в двоичной системе счисления.

Размерность целого двоичного числа может составлять 8, 16 или 32 бит. Знак двоичного числа определяется тем, как интерпретируется старший бит в представлении числа. Это 7, 15 или 31-й биты для чисел соответствующей размерности. При этом интересно то, что среди арифметических команд есть всего две команды, которые действительно учитывают этот старший разряд как знаковый, — это команды целочисленного умножения и деления imul и idiv. В остальных случаях ответственность за действия со знаковыми числами и, соответственно, со знаковым разрядом ложится на программиста. Диапазон значений двоичного числа зависит от его размера и трактовки старшего бита либо как старшего значащего бита числа, либо как бита знака числа (табл. 9).

Таблица 9

Диапазон значений двоичных чисел

Размерность поля	Целое без знака	Целое со знаком
байт	0 ... 255	–128 ... +127
слово	0 ... 65 535	–32 768 ... +32 767
двойное слово	0 ... 4 294 967 295	–2 147 483 648 ... +2 147 483 647

Десятичные числа

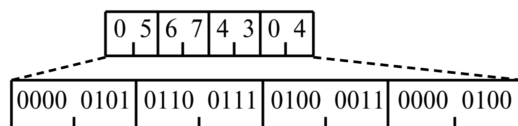
Десятичные числа — специальный вид представления числовой информации, в основу которого положен принцип кодирования каждой десятичной цифры числа группой из четырех бит. При этом каждый байт числа содержит одну или две десятичные цифры в так называемом двоично-десятичном коде (BCD — Binary-Coded Decimal). Микропроцессор хранит BCD-числа в двух форматах (рис. 28):

1) упакованном формате. В этом формате каждый байт содержит две десятичные цифры. Десятичная цифра представляет собой двоичное значение в диапазоне от 0 до 9 размером 4 би-

та. При этом код старшей цифры числа занимает старшие 4 бита. Следовательно, диапазон представления десятичного упакованного числа в 1 байте составляет от 00 до 99;

2) неупакованном формате. В этом формате каждый байт содержит одну десятичную цифру в четырех младших битах. Старшие 4 бита имеют нулевое значение. Это так называемая зона. Следовательно, диапазон представления десятичного неупакованного числа в 1 байте составляет от 0 до 9.

Упакованное десятичное число 5674304



Неупакованное десятичное число 9985784

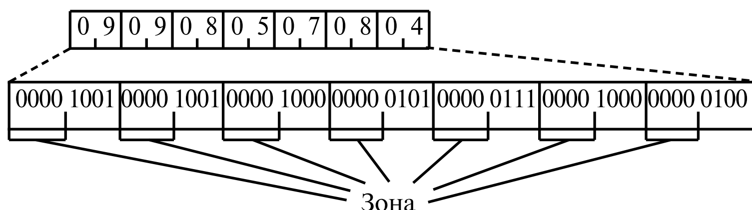


Рис. 28. Представление BCD-чисел

Как описать двоично-десятичные числа в программе? Для этого можно использовать только две директивы описания и инициализации данных — db и dt. Возможность применения только этих директив для описания BCD-чисел обусловлена тем, что к таким числам также применим принцип «младший байт по младшему адресу», что очень удобно для их обработки. И вообще, при использовании такого типа данных как BCD-числа, порядок описания этих чисел в программе и алгоритм их обработки — это дело вкуса и личных пристрастий программиста. Это станет ясно после того, как мы ниже рассмотрим основы работы с BCD-числами.

Арифметические операции над целыми двоичными числами

Сложение двоичных чисел без знака

Микропроцессор выполняет сложение операндов по правилам сложения двоичных чисел. Проблем не возникает до тех пор,

пока значение результата не превышает размерности поля операнда. Например, при сложении операндов размером в байт результат не должен превышать число 255. Если это происходит, то результат оказывается неверным. Рассмотрим, почему так происходит.

К примеру, выполним сложение: 254 + 5 = 259 в двоичном виде. 11111110 + 0000101 = 1 00000011. Результат вышел за пределы 8 бит и правильное его значение укладывается в 9 бит, а в 8-битовом поле операнда осталось значение 3, что, конечно, неверно. В микропроцессоре этот исход сложения прогнозируется и предусмотрены специальные средства для фиксирования подобных ситуаций и их обработки. Так, для фиксирования ситуации выхода за разрядную сетку результата, как в данном случае, предназначен флаг переноса cf. Он располагается в бите 0 регистра флагов EFLAGS/FLAGS. Именно установкой этого флага фиксируется факт переноса единицы из старшего разряда операнда. Естественно, что программист должен учитывать возможность такого исхода операции сложения и предусматривать средства для корректировки. Это предполагает включение участков кода после операции сложения, в которых анализируется флаг cf. Анализ этого флага можно провести различными способами.

Самый простой и доступный — использовать команду условного перехода ***jcc***. Эта команда в качестве операнда имеет имя метки в текущем сегменте кода. Переход на эту метку осуществляется в случае, если в результате работы предыдущей команды флаг cf установился в 1. В системе команд микропроцессора имеются три команды двоичного сложения:

1) **inc операнд** — операция инкремента, т. е. увеличения значения операнда на 1;
2) **add операнд_1, операнд_2** — команда сложения с принципом действия: операнд_1 = операнд_1 + операнд_2;
3) **adc операнд_1,операнд_2** — команда сложения с учетом флага переноса cf. Принцип действия команды: операнд_1 = операнд_1 + операнд_2 + значение_cf.

Обратите внимание на последнюю команду — это команда сложения, учитывающая перенос единицы из старшего разряда. Механизм появления такой единицы мы уже рассмотрели. Таким образом, команда adc является средством микропроцессора для сложения длинных двоичных чисел, размерность которых превосходит поддерживаемые микропроцессором длины стандартных полей.

Сложение двоичных чисел со знаком

На самом деле микропроцессор «не подозревает» о различии между числами со знаком и без знака. Вместо этого у него есть средства фиксирования возникновения характерных ситуаций, складывающихся в процессе вычислений. Некоторые из них мы рассмотрели при обсуждении сложения чисел без знака:

1) флаг переноса cf, установка которого в 1 говорит о том, что произошел выход за пределы разрядности операндов;
2) команду adc, которая учитывает возможность такого выхода (перенос из младшего разряда).

Другое средство — это регистрация состояния старшего (знакового) разряда операнда, которое осуществляется с помощью флага переполнения of в регистре EFLAGS (бит 11).

Вы, конечно, помните, как представляются числа в компьютере: положительные — в двоичном коде, отрицательные — в дополнительном коде. Рассмотрим различные варианты сложения чисел. Примеры призваны показать поведение двух старших битов операндов и правильность результата операции сложения.

Пример

30566 = 01110111 01100110

\+

00687 = 00000010 10101111

=

31253 = 01111010 00010101

Следим за переносами из 14 и 15-го разрядов и правильностью результата: переносов нет, результат правильный.

Пример

30566 = 01110111 01100110

\+

30566 = 01110111 01100110

=

1132 = 11101110 11001100

Произошел перенос из 14-го разряда; из 15-го разряда переноса нет. Результат неправильный, так как имеется переполнение — значение числа получилось больше, чем то, которое может иметь 16-битное число со знаком (+32 767).

Пример
–30566 = 10001000 10011010
+
–04875 = 11101100 11110101
=
–35441 = 01110101 10001111

Произошел перенос из 15-го разряда, из 14-го разряда нет переноса. Результат неправильный, так как вместо отрицательного числа получилось положительное (в старшем бите находится 0).

Пример
–4875 = 11101100 11110101
+
–4875 = 11101100 11110101
=
09750 = 11011001 11101010

Есть переносы из 14 и 15-го разрядов. Результат правильный.

Таким образом, мы исследовали все случаи и выяснили, что ситуация переполнения (установка флага OF в 1) происходит при переносе:

1) из 14-го разряда (для положительных чисел со знаком);
2) из 15-го разряда (для отрицательных чисел).

И наоборот, переполнения не происходит (т. е. флаг OF сбрасывается в 0), если есть перенос из обоих разрядов или перенос отсутствует в обоих разрядах.

Итак, переполнение регистрируется с помощью флага переполнения of. Дополнительно к флагу of при переносе из старшего разряда устанавливается в 1 и флаг переноса CF. Так как микропроцессор не знает о существовании чисел со знаком и без знака, то вся ответственность за правильность действий с получившимися числами ложится на программиста. Проанализировать флаги CF и OF можно командами условного перехода JC\JNC и JO\JNO соответственно.

Что же касается команд сложения чисел со знаком, то они те же, что и для чисел без знака.

Вычитание двоичных чисел без знака

Как и при анализе операции сложения, порассуждаем над сутью процессов, происходящих при выполнении операции вы-

читания. Если уменьшаемое больше вычитаемого, то проблем нет, — разность положительна, результат верен. Если уменьшаемое меньше вычитаемого, возникает проблема: результат меньше 0, а это уже число со знаком. В этом случае результат необходимо завернуть. Что это означает? При обычном вычитании (в столбик) делают заем 1 из старшего разряда. Микропроцессор поступает аналогично, т. е. занимает 1 из разряда, следующего за старшим, в разрядной сетке операнда. Поясним на примере.

Пример

05 = 00000000 00000101

–10 = 00000000 00001010

Для того чтобы произвести вычитание, произведем

воображаемый заем из старшего разряда:

100000000 00000101

–

00000000 00001010

=

11111111 11111011

Тем самым, по сути, выполняется действие

(65 536 + 5) — 10 = 65 531

0 здесь как бы эквивалентен числу 65536. Результат, конечно, неверен, но микропроцессор считает, что все нормально, хотя факт заема единицы он фиксирует установкой флага переноса cf. Но посмотрите еще раз внимательно на результат операции вычитания. Это же –5 в дополнительном коде! Проведем эксперимент: представим разность в виде суммы 5 + (–10).

Пример

5 = 00000000 00000101

+

(–10)= 11111111 11110110

=

11111111 11111011

т. е. мы получили тот же результат, что и в предыдущем примере.

Таким образом, после команды вычитания чисел без знака нужно анализировать состояние флага CF. Если он установлен в 1, то это говорит о том, что произошел заем из старшего разряда и результат получился в дополнительном коде.

Аналогично командам сложения группа команд вычитания состоит из минимально возможного набора. Эти команды выполняют вычитание по алгоритмам, которые мы сейчас рассматри-

ваем, а учет особых ситуаций должен производиться самим программистом. К командам вычитания относятся следующие:

1) dec операнд — операция декремента, т. е. уменьшения значения операнда на 1;

2) sub операнд_1,операнд_2 — команда вычитания; ее принцип действия: операнд_1 = операнд_1 — операнд_2;

3) sbb операнд_1,операнд_2 — команда вычитания с учетом заема (флага cf): операнд_1 = операнд_1 — операнд_2 — значение_cf.

Как видите, среди команд вычитания есть команда sbb, учитывающая флаг переноса cf. Эта команда подобна adc, но теперь уже флаг cf исполняет роль индикатора заема 1 из старшего разряда при вычитании чисел.

Вычитание двоичных чисел со знаком

Здесь все несколько сложнее. Микропроцессору незачем иметь два устройства — сложения и вычитания. Достаточно наличия только одного — устройства сложения. Но для вычитания способом сложения чисел со знаком в дополнительном коде необходимо представлять оба операнда — и уменьшаемое, и вычитаемое. Результат тоже нужно рассматривать как значение в дополнительном коде. Но здесь возникают сложности. Прежде всего они связаны с тем, что старший бит операнда рассматривается как знаковый. Рассмотрим пример вычитания 45 — (–127).

Пример

Вычитание чисел со знаком 1

45 = 0010 1101

–

–127 = 1000 0001

=

–44 = 1010 1100

Судя по знаковому разряду, результат получился отрицательный, что, в свою очередь, говорит о том, что число нужно рассматривать как дополнение, равное –44. Правильный результат должен быть равен 172. Здесь мы, как и в случае знакового сложения, встретились с переполнением мантиссы, когда значащий разряд числа изменил знаковый разряд операнда. Отследить такую ситуацию можно по содержимому флага переполнения of. Его установка в 1 говорит о том, что результат вышел за диапазон представления знаковых чисел (т. е. изменился старший бит) для операнда

данного размера, и программист должен предусмотреть действия по корректировке результата.

Пример

Вычитание чисел со знаком 2

–45 – 45 = –45 + (–45)= –90.

–45 = 1101 0011

+

–45 = 1101 0011

=

–90 = 1010 0110

Здесь все нормально, флаг переполнения of сброшен в 0, а 1 в знаковом разряде говорит о том, что значение результата — число в дополнительном коде.

Вычитание и сложение операндов большой размерности

Если вы заметили, команды сложения и вычитания работают с операндами фиксированной размерности: 8, 16, 32 бит. А что делать, если нужно сложить числа большей размерности, например 48 бит, используя 16-разрядные операнды? К примеру, сложим два 48-разрядных числа:

1 слагаемое **0010001110010101 1001011111111000 1111111100110001**
+
2 слагаемое **0100010010001011 1010010100100100 0100100110000110**

1 шаг: сложение младших 16 бит 10100100010110111
перенос в старший разряд

2 шаг: сложение средних 16 бит **0100101111111000**
(с учетом переноса из младшего разряда)
+
1010010100100100
1111000100011100
0000000000000001
1111000100011101

3 шаг: сложение старших 16 бит
(переноса из младшего разряда нет)

0010001110010101
+
0100010010001011

Результат сложения **0110100000100000 1111000100011101 0100100010110111**

Рис. 29. Сложение операндов большой размерности

На рисунке 29 по шагам показана технология сложения длинных чисел. Видно, что процесс сложения многобайтных чисел происходит так же, как и при сложении двух чисел «в столбик», — с осуществлением при необходимости переноса 1 в старший разряд. Если нам удастся запрограммировать этот процесс, то мы значительно расширим диапазон двоичных чисел, над которыми мы сможем выполнять операции сложения и вычитания.

Принцип вычитания чисел с диапазоном представления, превышающим стандартные разрядные сетки операндов, тот же, что и при сложении, т. е. используется флаг переноса cf. Нужно только представлять себе процесс вычитания в столбик и правильно комбинировать команды микропроцессора с командой sbb.

В завершение обсуждения команд сложения и вычитания отметим, что кроме флагов cf и of в регистре eflags есть еще несколько флагов, которые можно использовать с двоичными арифметическими командами. Речь идет о следующих флагах:

1) zf — флаг нуля, который устанавливается в 1, если результат операции равен 0, и в 1, если результат не равен 0;
2) sf — флаг знака, значение которого после арифметических операций (и не только) совпадает со значением старшего бита результата, т. е. с битом 7, 15 или 31. Таким образом, этот флаг можно использовать для операций над числами со знаком.

Умножение чисел без знака

Для умножения чисел без знака предназначена команда
mul сомножитель_1

Как видите, в команде указан всего лишь один операнд-сомножитель. Второй операнд-сомножитель_2 задан неявно. Его местоположение фиксировано и зависит от размера сомножителей. Так как в общем случае результат умножения больше, чем любой из его сомножителей, то его размер и местоположение должны быть тоже определены однозначно. Варианты размеров сомножителей и размещения второго операнда и результата приведены в таблице 10.

Расположение операндов и результата при умножении *Таблица 10*

Сомножитель_1	Сомножитель_2	Результат
Байт	al	16 бит в ax: al — младшая часть результата; ah — старшая часть результата
Слово	ax	32 бит в паре dx:ax: ax — младшая часть результата; dx — старшая часть результата

Окончание табл. 10

Сомножитель_1	Сомножитель_2	Результат
Двойное слово	eax	64 бит в паре edx:eax: eax — младшая часть результата; edx — старшая часть результата

Из таблицы видно, что произведение состоит из двух частей и в зависимости от размера операндов размещается в двух местах — на месте сомножитель_2 (младшая часть) и в дополнительном регистре ah, dx, edx (старшая часть). Как же динамически (т. е. во время выполнения программы) узнать, что результат достаточно мал и уместился в одном регистре или что он превысил размерность регистра и старшая часть оказалась в другом регистре? Для этого привлекаются уже известные нам по предыдущему обсуждению флаги переноса cf и переполнения of:

1) если старшая часть результата нулевая, то после операции произведения флаги cf = 0 и of = 0;
2) если же эти флаги ненулевые, то это означает, что результат вышел за пределы младшей части произведения и состоит из двух частей, что и нужно учитывать при дальнейшей работе.

Умножение чисел со знаком

Для умножения чисел со знаком предназначена команда
[imul операнд_1,операнд_2,операнд_3]

Эта команда выполняется так же, как и команда mul. Отличительной особенностью команды imul является только формирование знака.

Если результат мал и умещается в одном регистре (т. е. если cf = of = 0), то содержимое другого регистра (старшей части) является расширением знака — все его биты равны старшему биту (знаковому разряду) младшей части результата. В противном случае (если cf = of = 1) знаком результата является знаковый бит старшей части результата, а знаковый бит младшей части является значащим битом двоичного кода результата.

Деление чисел без знака

Для деления чисел без знака предназначена команда
div делитель

Делитель может находиться в памяти или в регистре и иметь размер 8, 16 или 32 бит. Местонахождение делимого фиксировано и так же, как в команде умножения, зависит от размера операндов. Результатом команды деления являются значения частного и остатка.

Варианты местоположения и размеров операндов операции деления показаны в таблице 11.

Расположение операндов и результата при делении *Таблица 11*

Делимое	**Делитель**	**Частное**	**Остаток**
16 бит в регистре ax	Байт регистр или ячейка памяти	Байт в регистре al	Байт в регистре ah
32 бит dx — старшая часть, ax — младшая часть	Слово 16 бит регистр или ячейка памяти	Слово 16 бит в регистре ax	Слово 16 бит в регистре dx
64 бит edx — старшая часть, eax — младшая часть	Двойное слово 32 бит регистр или ячейка памяти	Двойное слово 32 бит в регистре eax	Двойное слово 32 бит в регистре edx

После выполнения команды деления содержимое флагов неопределенно, но возможно возникновение прерывания с номером 0, называемого «деление на нуль». Этот вид прерывания относится к так называемым исключениям. Эта разновидность прерываний возникает внутри микропроцессора из-за некоторых аномалий во время вычислительного процесса. Прерывание 0, «деление на нуль», при выполнении команды div может возникнуть по одной из следующих причин:

1) делитель равен нулю;
2) частное не входит в отведенную под него разрядную сетку, что может произойти в следующих случаях:
   а) при делении делимого величиной в слово на делитель величиной в байт, причем значение делимого в более чем 256 раз больше значения делителя;
   б) при делении делимого величиной в двойное слово на делитель величиной в слово, причем значение делимого в более чем 65 536 раз больше значения делителя;

в) при делении делимого величиной в учетверенное слово на делитель величиной в двойное слово, причем значение делимого в более чем 4 294 967 296 раз больше значения делителя.

Деление чисел со знаком

Для деления чисел со знаком предназначена команда
idiv делитель
Для этой команды справедливы все рассмотренные положения, касающиеся команд и чисел со знаком. Отметим лишь особенности возникновения исключения 0, «деление на нуль», в случае чисел со знаком. Оно возникает при выполнении команды idiv по одной из следующих причин:

1) делитель равен нулю;
2) частное не входит в отведенную для него разрядную сетку.

Последнее в свою очередь может произойти:

1) при делении делимого величиной в слово со знаком на делитель величиной в байт со знаком, причем значение делимого в более чем 128 раз больше значения делителя (таким образом, частное не должно находиться вне диапазона от –128 до +127);
2) при делении делимого величиной в двойное слово со знаком на делитель величиной в слово со знаком, причем значение делимого в более чем 32 768 раз больше значения делителя (таким образом, частное не должно находиться вне диапазона от –32 768 до +32 768);
3) при делении делимого величиной в учетверенное слово со знаком на делитель величиной в двойное слово со знаком, причем значение делимого в более чем 2 147 483 648 раз больше значения делителя (таким образом, частное не должно находиться вне диапазона от –2 147 483 648 до +2 147 483 647).

Вспомогательные команды для целочисленных операций

В системе команд микропроцессора есть несколько команд, которые могут облегчить программирование алгоритмов, производящих арифметические вычисления. В них могут возникать различные проблемы, для разрешения которых разработчики микропроцессора предусмотрели несколько команд.

Команды преобразования типов

Что делать, если размеры операндов, участвующих в арифметических операциях, разные? Например, предположим, что в операции сложения один операнд является словом, а другой занимает двойное слово. Выше сказано, что в операции сложения должны участвовать операнды одного формата. Если числа без знака, то выход найти просто. В этом случае можно на базе исходного операнда сформировать новый (формата двойного слова), старшие разряды которого просто заполнить нулями. Сложнее ситуация для чисел со знаком: как динамически, в ходе выполнения программы, учесть знак операнда? Для решения подобных проблем в системе команд микропроцессора есть так называемые команды преобразования типа. Эти команды расширяют байты в слова, слова — в двойные слова и двойные слова — в учетверенные слова (64-разрядные значения). Команды преобразования типа особенно полезны при преобразовании целых со знаком, так как они автоматически заполняют старшие биты вновь формируемого операнда значениями знакового бита старого объекта. Эта операция приводит к целым значениям того же знака и той же величины, что и исходная, но уже в более длинном формате. Подобное преобразование называется *операцией распространения знака*.

Существуют два вида команд преобразования типа.

1. Команды без операндов. Эти команды работают с фиксированными регистрами:

   1) cbw (Convert Byte to Word) — команда преобразования байта (в регистре al) в слово (в регистре ax) путем распространения значения старшего бита al на все биты регистра ah;

   2) cwd (Convert Word to Double) — команда преобразования слова (в регистре ax) в двойное слово (в регистрах dx:ax) путем распространения значения старшего бита ax на все биты регистра dx;

   3) cwde (Convert Word to Double) — команда преобразования слова (в регистре ax) в двойное слово (в регистре eax) путем распространения значения старшего бита ax на все биты старшей половины регистра eax;

   4) cdq (Convert Double Word to Quarter Word) — команда преобразования двойного слова (в регистре eax) в учетверенное слово (в регистрах edx:eax) путем распространения значения старшего бита eax на все биты регистра edx.

2. Команды movsx и movzx, относящиеся к командам обработки строк. Эти команды обладают полезным свойством в контексте нашей проблемы:

1) movsx операнд_1,операнд_2 — переслать с распространением знака. Расширяет 8 или 16-разрядное значение операнд_2, которое может быть регистром или операндом в памяти, до 16 или 32-разрядного значения в одном из регистров, используя значение знакового бита для заполнения старших позиций операнд_1. Данную команду удобно использовать для подготовки операндов со знаками к выполнению арифметических действий;

2) movzx операнд_1,операнд_2 — переслать с расширением нулем. Расширяет 8- или 16-разрядное значение операнд_2 до 16- или 32-разрядного с очисткой (заполнением) нулями старших позиций операнд_2. Данную команду удобно использовать для подготовки операндов без знака к выполнению арифметических действий.

Другие полезные команды

1. **xadd назначение, источник** — обмен местами и сложение. Команда позволяет выполнить последовательно два действия:

1) обменять значения *назначение и источник*;

2) поместить на место операнда *назначение* сумму:

назначение = назначение + источник.

2. **neg операнд** — отрицание с дополнением до двух.

Команда выполняет инвертирование значения операнд. Физически команда выполняет одно действие:

операнд = 0 — операнд, т. е. вычитает операнд из нуля.

Команду neg операнд можно применять:

1) для смены знака;

2) для выполнения вычитания из константы.

Арифметические операции над двоично-десятичными числами

В данном разделе мы рассмотрим особенности каждого из четырех основных арифметических действий для упакованных и неупакованных двоично-десятичных чисел.

Справедливо может возникнуть вопрос: а зачем нужны BCD-числа? Ответ может быть следующим: BCD-числа нужны в деловых приложениях, т. е. там, где числа должны быть большими

и точными. Как мы уже убедились на примере двоичных чисел, операции с такими числами довольно проблематичны для языка ассемблера. К недостаткам использования двоичных чисел можно отнести следующие:

1) значения величин в формате слова и двойного слова имеют ограниченный диапазон. Если программа предназначена для работы в области финансов, то ограничение суммы в рублях величиной 65 536 (для слова) или даже 4 294 967 296 (для двойного слова) будет существенно сужать сферу ее применения;
2) наличие ошибок округления. Представляете себе программу, работающую где-нибудь в банке, которая не учитывает величину остатка при действиях с целыми двоичными числами и оперирует при этом миллиардами? Не хотелось бы быть автором такой программы. Применение чисел с плавающей точкой не спасет — там существует та же проблема округления;
3) представление большого объема результатов в символьном виде (ASCII-коде). Деловые программы не просто выполняют вычисления; одной из целей их использования является оперативная выдача информации пользователю. Для этого, естественно, информация должна быть представлена в символьном виде. Перевод чисел из двоичного кода в ASCII-код требует определенных вычислительных затрат. Число с плавающей точкой еще труднее перевести в символьный вид. А вот если посмотреть на шестнадцатеричное представление неупакованной десятичной цифры и на соответствующий ей символ в таблице ASCII, то видно, что они отличаются на величину 30h. Таким образом, преобразование в символьный вид и обратно получается намного проще и быстрее.

Наверное вы уже убедились в важности овладения хотя бы основами действий с десятичными числами. Далее рассмотрим особенности выполнения основных арифметических операций с десятичными числами. Отметим сразу тот факт, что отдельных команд сложения, вычитания, умножения и деления BCD-чисел нет. Сделано это по вполне понятным причинам: размерность таких чисел может быть сколь угодно большой. Складывать и вычитать можно двоично-десятичные числа, как в упакованном формате, так и в неупакованном, а вот делить и умножать можно только неупакованные BCD-числа. Почему это так, будет видно из дальнейшего обсуждения.

Сложение неупакованных BCD-чисел

Рассмотрим два случая сложения.

Пример

Результат сложения не больше 9

6 = 0000 0110

+

3 = 0000 0011

=

9 = 0000 1001

Переноса из младшей тетрады в старшую нет. Результат правильный.

Пример

Результат сложения больше 9:

06 = 0000 0110

+

07 = 0000 0111

=

13 = 0000 1101

Мы получили уже не BCD-число. Результат неправильный. Правильный результат в неупакованном BCD-формате должен быть таким: 0000 0001 0000 0011 в двоичном представлении (или 13 в десятичном).

Проанализировав данную проблему при сложении BCD-чисел (и подобные проблемы при выполнении других арифметических действий) и возможные пути ее решения, разработчики системы команд микропроцессора решили не вводить специальные команды для работы с BCD-числами, а ввести несколько корректировочных команд.

Назначение этих команд — в корректировке результата работы обычных арифметических команд для случаев, когда операнды в них являются BCD-числами.

В случае вычитания в примере 10 видно, что полученный результат нужно корректировать. Для коррекции операции сложения двух однозначных неупакованных BCD-чисел в системе команд микропроцессора существует специальная команда — **aaa (ASCII Adjust for Addition)** — коррекция результата сложения для представления в символьном виде.

Эта команда не имеет операндов. Она работает неявно только с регистром al и анализирует значение его младшей тетрады:

1) если это значение меньше 9, то флаг cf сбрасывается в 0 и осуществляется переход к следующей команде;

2) если это значение больше 9, то выполняются следующие действия:

а) к содержимому младшей тетрады al (но не к содержимому всего регистра!) прибавляется 6, тем самым значение десятичного результата корректируется в правильную сторону;

б) флаг cf устанавливается в 1, тем самым фиксируется перенос в старший разряд, для того чтобы его можно было учесть в последующих действиях.

Так, в примере 10, предполагая, что значение суммы 0000 1101 находится в al, после команды aaa в регистре будет 1101 + 0110 = = 0011, т. е. двоичное 0000 0011 или десятичное 3, а флаг cf установится в 1, т. е. перенос запомнился в микропроцессоре. Далее программисту нужно будет использовать команду сложения adc, которая учтет перенос из предыдущего разряда.

Вычитание неупакованных BCD-чисел

Ситуация здесь вполне аналогична сложению. Рассмотрим те же случаи.

Пример

Результат вычитания не больше 9:

6 = 0000 0110

—

3 = 0000 0011

=

3 = 0000 0011

Как видим, заема из старшей тетрады нет. Результат верный и корректировки не требует.

Пример

Результат вычитания больше 9:

6 = 0000 0110

–

7 = 0000 0111

=

–1 = 1111 1111

Вычитание проводится по правилам двоичной арифметики. Поэтому результат не является BCD-числом.

Правильный результат в неупакованном BCD-формате должен быть 9 (0000 1001 в двоичной системе счисления). При этом предполагается заем из старшего разряда, как при обычной команде вычитания, т. е. в случае с BCD числами фактически должно быть выполнено вычитание 16 — 7. Таким образом, видно: как и в случае сложения, результат вычитания нужно корректировать. Для этого существует специальная команда — **aas (ASCII Adjust for Substraction)** — коррекция результата вычитания для представления в символьном виде.

Команда aas также не имеет операндов и работает с регистром al, анализируя его младшую тетраду следующим образом:

1) если ее значение меньше 9, то флаг cf сбрасывается в 0 и управление передается следующей команде;
2) если значение тетрады в al больше 9, то команда aas выполняет следующие действия:
   а) из содержимого младшей тетрады регистра al (заметьте — не из содержимого всего регистра) вычитает 6;
   б) обнуляет старшую тетраду регистра al;
   в) устанавливает флаг cf в 1, тем самым фиксируя воображаемый заем из старшего разряда.

Понятно, что команда aas применяется вместе с основными командами вычитания sub и sbb. При этом команду sub есть смысл использовать только один раз, при вычитании самых младших цифр операндов, далее должна применяться команда sbb, которая будет учитывать возможный заем из старшего разряда.

Умножение неупакованных BCD-чисел

На примере сложения и вычитания неупакованных чисел стало понятно, что стандартных алгоритмов для выполнения этих действий над BCD-числами нет и программист должен сам, исходя из требований к своей программе, реализовать эти операции.

Реализация двух оставшихся операций — умножения и деления — еще более сложна. В системе команд микропроцессора присутствуют только средства для производства умножения и деления одноразрядных неупакованных BCD-чисел.

Для того чтобы умножать числа произвольной размерности, нужно реализовать процесс умножения самостоятельно, взяв за основу некоторый алгоритм умножения, например «в столбик».

Для того чтобы перемножить два одноразрядных BCD-числа, необходимо:

1) поместить один из сомножителей в регистр AL (как того требует команда mul);
2) поместить второй операнд в регистр или память, отведя байт;
3) перемножить сомножители командой mul (результат, как и положено, будет в ах);
4) результат, конечно, получится в двоичном коде, поэтому его нужно скорректировать.

Для коррекции результата после умножения применяется специальная команда — **aam (ASCII Adjust for Multiplication)** — коррекция результата умножения для представления в символьном виде.

Она не имеет операндов и работает с регистром AX следующим образом:

1) делит al на 10;
2) результат деления записывается так: частное в al, остаток в ah.

В результате после выполнения команды aam в регистрах AL и ah находятся правильные двоично-десятичные цифры произведения двух цифр.

Перед окончанием обсуждения команды aam необходимо отметить еще один вариант ее применения. Эту команду можно применять для преобразования двоичного числа в регистре AL в неупакованное BCD-число, которое будет размещено в регистре ax: старшая цифра результата в ah, младшая — в al. Понятно, что двоичное число должно быть в диапазоне 0 ... 99.

Деление неупакованных BCD-чисел

Процесс выполнения операции деления двух неупакованных BCD-чисел несколько отличается от других, рассмотренных ранее операций с ними. Здесь также требуются действия по коррекции, но они должны осуществляться до основной операции, выполняющей непосредственно деление одного BCD-числа на другое BCD-число. Предварительно в регистре ax нужно получить две неупакованные BCD-цифры делимого. Это делает программист удобным для него способом. Далее нужно выдать команду aad — **aad (ASCII Adjust for Division)** — коррекция деления для представления в символьном виде.

Команда не имеет операндов и преобразует двузначное неупакованное BCD-число в регистре ax в двоичное число. Это двоичное число впоследствии будет играть роль делимого в операции деления. Кроме преобразования, команда aad помещает полученное двоичное число в регистр AL. Делимое, естественно, будет двоичным числом из диапазона 0 ... 99.

Алгоритм, по которому команда aad осуществляет это преобразование, состоит в следующем:

1) умножить старшую цифру исходного BCD-числа в ax (содержимое AH) на 10;
2) выполнить сложение AH + AL, результат которого (двоичное число) занести в AL;
3) обнулить содержимое AH.

Далее программисту нужно выдать обычную команду деления div для выполнения деления содержимого ax на одну BCD-цифру, находящуюся в байтовом регистре или байтовой ячейке памяти.

Аналогично aam, команде aad можно найти и другое применение — использовать ее для перевода неупакованных BCD-чисел из диапазона 0 ... 99 в их двоичный эквивалент.

Для деления чисел большей разрядности, так же как и в случае умножения, нужно реализовывать свой алгоритм, например «в столбик», либо найти более оптимальный путь.

Арифметические действия над упакованными BCD-числами

Как уже отмечалось выше, упакованные BCD-числа можно только складывать и вычитать. Для выполнения других действий над ними их нужно дополнительно преобразовывать либо в неупакованный формат, либо в двоичное представление. Из-за того, что упакованные BCD-числа представляют не слишком большой интерес, мы их рассмотрим кратко.

Сложение упакованных BCD-чисел

Вначале разберемся с сутью проблемы и попытаемся сложить два двузначных упакованных BCD-числа.

Пример

Сложение упакованных BCD-чисел:

67 = 0110 0111

\+

+

75 = 0111 0101

=

142 = 1101 1100 = 220

Как видим, в двоичном виде результат равен 1101 1100 (или 220 в десятичном представлении), что неверно. Это происходит по той причине, что микропроцессор не подозревает о существовании BCD-чисел и складывает их по правилам сложения двоичных чисел. На самом деле, результат в двоично-десятичном виде должен быть равен 0001 0100 0010 (или 142 в десятичном представлении).

Видно, что, как и для неупакованных BCD-чисел, для упакованных BCD-чисел существует потребность как-то корректировать результаты арифметических операций.

Микропроцессор предоставляет для этого команду daa — **daa (Decimal Adjust for Addition)** — коррекция результата сложения для представления в десятичном виде.

Команда daa преобразует содержимое регистра al в две упакованные десятичные цифры по алгоритму, приведенному в описании команды daa. Получившаяся в результате сложения единица (если результат сложения больше 99) запоминается в флаге cf, тем самым учитывается перенос в старший разряд.

Вычитание упакованных BCD-чисел

Аналогично сложению, микропроцессор рассматривает упакованные BCD-числа как двоичные и, соответственно, выполняет вычитание BCD-чисел как двоичных.

Пример

Вычитание упакованных BCD-чисел.

Выполним вычитание 67—75. Так как микропроцессор выполняет вычитание способом сложения, то и мы последуем этому:

67 = 0110 0111

+

–75 = 1011 0101

=

–8 = 0001 1100 = 28

Как видим, результат равен 28 в десятичной системе счисления, что является абсурдом. В двоично-десятичном коде резуль-

тат должен быть равен 0000 1000 (или 8 в десятичной системе счисления).

При программировании вычитания упакованных BCD-чисел программист, как и при вычитании неупакованных BCD-чисел, должен сам осуществлять контроль за знаком. Это делается с помощью флага CF, который фиксирует заем из старших разрядов.

Само вычитание BCD-чисел осуществляется простой командой вычитания sub или sbb. Коррекция результата осуществляется командой das — **das (Decimal Adjust for Substraction)** — коррекция результата вычитания для представления в десятичном виде.

Команда das преобразует содержимое регистра AL в две упакованные десятичные цифры по алгоритму, приведенному в описании команды das.

ЛЕКЦИЯ № 19. Команды передачи управления

1. Логические команды

Наряду со средствами арифметических вычислений, система команд микропроцессора имеет также средства логического преобразования данных. Под логическими понимаются такие преобразования данных, в основе которых лежат *правила формальной логики*.

Формальная логика работает на уровне утверждений **истинно** и **ложно**. Для микропроцессора это, как правило, означает **1** и **0** соответственно. Для компьютера язык нулей и единиц является родным, но минимальной единицей данных, с которой работают машинные команды, является байт. Однако на системном уровне часто необходимо иметь возможность работать на предельно низком уровне — на уровне бит.

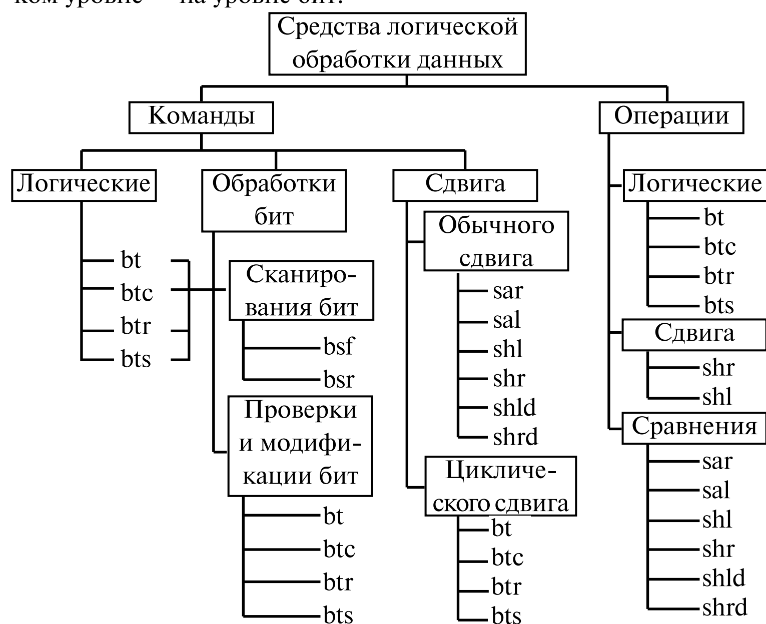


Рис. 29. Средства логической обработки данных

К средствам логического преобразования данных относятся логические команды и логические операции. Операнд команды ассемблера в общем случае может представлять собой выражение, которое, в свою очередь, является комбинаций операторов и операндов. Среди этих операторов могут быть и операторы, реализующие логические операции над объектами выражения.

Перед подробным рассмотрением этих средств рассмотрим, что же представляют собой сами логические данные и какие операции над ними производятся.

Логические данные

Теоретической базой для логической обработки данных является *формальная логика*. Существует несколько систем логики. Одна из наиболее известных — это *исчисление высказываний*. **Высказывание** — это любое утверждение, о котором можно сказать, что оно либо *истинно*, либо *ложно*.

Исчисление высказываний представляет собой совокупность правил, используемых для определения истинности или ложности некоторой комбинации высказываний.

Исчисление высказываний очень гармонично сочетается с принципами работы компьютера и основными методами его программирования. Все аппаратные компоненты компьютера построены на логических микросхемах. Система представления информации в компьютере на самом нижнем уровне основана на понятии бита. Бит, имея всего два состояния (0 (ложно) и 1 (истинно)), естественным образом вписывается в исчисление высказываний.

Согласно теории, над высказываниями (над битами) могут выполняться следующие логические операции.

1.**Отрицание** (логическое *НЕ*) — логическая операция над одним операндом, результатом которой является величина, обратная значению исходного операнда.

Эта операция однозначно характеризуется следующей таблицей истинности (табл. 12).

Таблица истинности для логического отрицания

Таблица 12

Значение операнда	0	1
Результат операции	1	0

2. **Логическое сложение** (логическое включающее *ИЛИ*) — логическая операция над двумя операндами, результатом которой является «истина» (1), если один или оба операнда имеют значение «истина» (1), и «ложь» (0), если оба операнда имеют значение «ложь» (0).

Эта операция описывается с помощью следующей таблицы истинности (табл. 13).

Таблица истинности для логического включающего ИЛИ *Таблица 13*

Значение операнда 1	0	0	1	1
Значение операнда 2	0	1	0	1
Результат операции	0	1	1	1

3. **Логическое умножение** (логическое *И*) — логическая операция над двумя операндами, результатом которой является «истина» (1) только в том случае, если оба операнда имеют значение «истина» (1). Во всех остальных случаях значение операции «ложь» (0).

Эта операция описывается с помощью следующей таблицы истинности (табл. 14).

Таблица истинности для логического И *Таблица 14*

Значение операнда 1	0	0	1	1
Значение операнда 2	0	1	0	1
Результат операции	0	0	0	1

4. **Логическое исключающее сложение** (логическое исключающее *ИЛИ*) — логическая операция над двумя операндами, результатом которой является «истина» (1), если только один из двух операндов имеет значение «истина» (1), и ложь (0), если оба операнда имеют значение «ложь» (0) или «истина» (1). Эта операция описывается с помощью следующей таблицы истинности (таб. 15).

Таблица истинности для логического исключающего ИЛИ
Таблица 15

Значение операнда 1	0	0	1	1
Значение операнда 2	0	1	0	1
Результат операции	0	1	1	0

Система команд микропроцессора содержит пять команд, поддерживающих данные операции. Эти команды выполняют логические операции над битами операндов. Размерность операндов, естественно, должна быть одинакова. Например, если размерность операндов равна слову (16 бит), то логическая операция выполняется сначала над нулевыми битами операндов, и ее результат записывается на место бита 0 результата. Далее команда последовательно повторяет эти действия над всеми битами с первого до пятнадцатого.

Логические команды

В системе команд микропроцессора есть следующий набор команд, поддерживающих работу с логическими данными:

1) **and операнд_1, операнд_2** — операция логического умножения. Команда выполняет поразрядно логическую операцию И (конъюнкцию) над битами операндов операнд_1 и операнд_2. Результат записывается на место операнд_1;
2) **or операнд_1, операнд_2** — операция логического сложения. Команда выполняет поразрядно логическую операцию ИЛИ (дизъюнкцию) над битами операндов операнд_1 и операнд_2. Результат записывается на место операнд_1;
3) **xor операнд_1, операнд_2** — операция логического исключающего сложения. Команда выполняет поразрядно логическую операцию исключающего ИЛИ над битами операндов операнд_1 и операнд_2. Результат записывается на место операнд_1;
4) **test операнд_1, операнд_2** — операция «проверить» (способом логического умножения). Команда выполняет поразрядно логическую операцию И над битами операндов операнд_1 и операнд_2. Состояние операндов остается прежним, изменяются только флаги zf, sf, и pf, что дает возможность анализировать состояние отдельных битов операнда без изменения их состояния;
5) **not операнд** — операция логического отрицания. Команда выполняет поразрядное инвертирование (замену значения на обратное) каждого бита операнда. Результат записывается на место операнда.

Для представления роли логических команд в системе команд микропроцессора очень важно понять области их применения и типовые приемы их использования при программировании.

С помощью логических команд возможно *выделение* отдельных битов в операнде с целью их *установки, сброса, инвертирования или просто проверки на определенное значение.*

Для организации подобной работы с битами операнд_2 обычно играет роль **маски**. С помощью установленных в 1 бите этой маски и определяются нужные для конкретной операции биты операнд_1. Покажем, какие логические команды могут применяться для этой цели:

1) для **установки** определенных разрядов (бит) в **1** применяется команда **or** операнд_1, операнд_2.

В этой команде операнд_2, выполняющий роль маски, должен содержать единичные биты на месте тех разрядов, которые должны быть установлены в 1 в операнд_1;

2) для **сброса** определенных разрядов (бит) в 0 применяется команда **and** операнд_1, операнд_2.

В этой команде операнд_2, выполняющий роль маски, должен содержать нулевые биты на месте тех разрядов, которые должны быть установлены в 0 в операнд_1;

3) команда **xor** операнд_1, операнд_2 применяется:

а) для выяснения того, какие биты в операнд_1 и операнд_2 **различаются**;

б) для **инвертирования** состояния заданных бит в операнд_1.

Интересующие нас биты маски (операнд_2) при выполнении команды xor должны быть единичными, остальные — нулевыми;

Для проверки состояния заданных бит применяется команда **test** операнд_1, операнд_2 (проверить операнд_1).

Проверяемые биты операнд_1 в маске (операнд_2) должны иметь единичное значение. Алгоритм работы команды test подобен алгоритму команды and, но он не меняет значения операнд_1. Результатом команды является установка значения флага нуля **zf:**

1) если zf = 0, то в результате логического умножения получился нулевой результат, т. е. один единичный бит маски, который **не совпал** с соответствующим единичным битом операнд_1;

2) если zf = 1, то в результате логического умножения получился ненулевой результат, т. е. *хотя бы один* единичный бит маски **совпал** с соответствующим единичным битом операнд_1.

Для реакции на результат команды test целесообразно использовать команду перехода **jnz** метка (Jump if Not Zero) — переход, если флаг нуля zf ненулевой, или команду с обратным действием — **jz** метка (Jump if Zero) — переход, если флаг нуля zf = 0.

Следующие две команды позволяют осуществить поиск первого установленного в 1 бита операнда. Поиск можно произвести как с начала, так и от конца операнда:

1) **bsf** операнд_1, операнд_2 (Bit Scaning Forward) — сканирование битов вперед. Команда просматривает (сканирует) биты операнд_2 от младшего к старшему (от бита 0 до старшего бита) в поисках первого бита, установленного в 1. Если таковой обнаруживается, в операнд_1 заносится номер этого бита в виде целочисленного значения. Если все биты операнд_2 равны 0, то флаг нуля zf устанавливается в 1, в противном случае флаг zf сбрасывается в 0;
2) **bsr** операнд_1, операнд_2 (Bit Scaning Reset) — сканирование битов в обратном порядке. Команда просматривает (сканирует) биты операнд_2 от старшего к младшему (от старшего бита к биту 0) в поисках первого бита, установленного в 1. Если таковой обнаруживается, в операнд_1 заносится номер этого бита в виде целочисленного значения. При этом важно, что позиция первого единичного бита слева отсчитывается все равно относительно бита 0. Если все биты операнд_2 равны 0, то флаг нуля zf устанавливается в 1, в противном случае флаг zf сбрасывается в 0.

В последних моделях микропроцессоров Intel в группе логических команд появилось еще несколько команд, которые позволяют осуществить доступ к одному конкретному биту операнда. Операнд может находиться как в памяти, так и в регистре общего назначения. Положение бита задается смещением бита относительно младшего бита операнда. Значение смещения может задаваться как в виде непосредственного значения, так и содержаться в регистре общего назначения. В качестве значения смещения вы можете использовать результаты работы команд bsr и bsf. Все команды присваивают значение выбранного бита флагу CF.

1) **bt операнд, смещение_бита** (Bit Test) — проверка бита. Команда переносит значение бита в флаг cf;
2) **bts операнд, смещение_бита** (Bit Test and Set) — проверка и установка бита. Команда переносит значение бита в флаг CF и затем устанавливает проверяемый бит в 1;

3) **btr операнд, смещение_бита** (Bit Test and Reset) — проверка и сброс бита. Команда переносит значение бита в флаг CF и затем устанавливает этот бит в 0;

4) **btc операнд, смещение_бита** (Bit Test and Convert) — проверка и инвертирование бита. Команда переносит значение бита в флаг cf и затем инвертирует значение этого бита.

Команды сдвига

Команды этой группы также обеспечивают манипуляции над отдельными битами операндов, но иным способом, чем логические команды, рассмотренные выше.

Все команды сдвига перемещают биты в поле операнда влево или вправо в зависимости от кода операции. Все команды сдвига имеют одинаковую структуру — **коп операнд, счетчик_сдвигов**.

Количество сдвигаемых разрядов — *счетчик_сдвигов* — располагается на месте второго операнда и может задаваться двумя способами:

1) *статически*, что предполагает задание фиксированного значения с помощью непосредственного операнда;

2) *динамически*, что означает занесение значения счетчика сдвигов в регистр cl перед выполнением команды сдвига.

Исходя из размерности регистра cl понятно, что значение счетчика сдвигов может лежать в диапазоне от 0 до 255. Но на самом деле это не совсем так. В целях оптимизации микропроцессор воспринимает только значение *пяти младших битов* счетчика, т. е. значение лежит в диапазоне от 0 до 31.

Все команды сдвига устанавливают флаг переноса **cf**.

По мере сдвига битов за пределы операнда они сначала попадают на флаг переноса, устанавливая его равным значению очередного бита, оказавшегося за пределами операнда. Куда этот бит попадет дальше, зависит от типа команды сдвига и алгоритма программы.

По принципу действия команды сдвига можно разделить на два типа:

1) команды линейного сдвига;
2) команды циклического сдвига.

Команды линейного сдвига

К командам этого типа относятся команды, осуществляющие сдвиг по следующему алгоритму:

1) очередной «выдвигаемый» бит устанавливает флаг CF;

2) бит, вводимый в операнд с другого конца, имеет значение 0;

3) при сдвиге очередного бита он переходит во флаг CF, при этом значение предыдущего сдвинутого бита теряется!

Команды линейного сдвига делятся на два подтипа:

1) команды логического линейного сдвига;

2) команды арифметического линейного сдвига.

К командам логического линейного сдвига относятся следующие:

1) **shl** операнд, счетчик_сдвигов (Shift Logical Left) — логический сдвиг влево. Содержимое операнда сдвигается влево на количество битов, определяемое значением счетчик_сдвигов. Справа (в позицию младшего бита) вписываются нули;

2) **shr** операнд, счетчик_сдвигов (Shift Logical Right) — логический сдвиг вправо. Содержимое операнда сдвигается вправо на количество битов, определяемое значением счетчик_сдвигов. Слева (в позицию старшего, знакового бита) вписываются нули.

На рисунке 30 показан принцип работы этих команд.

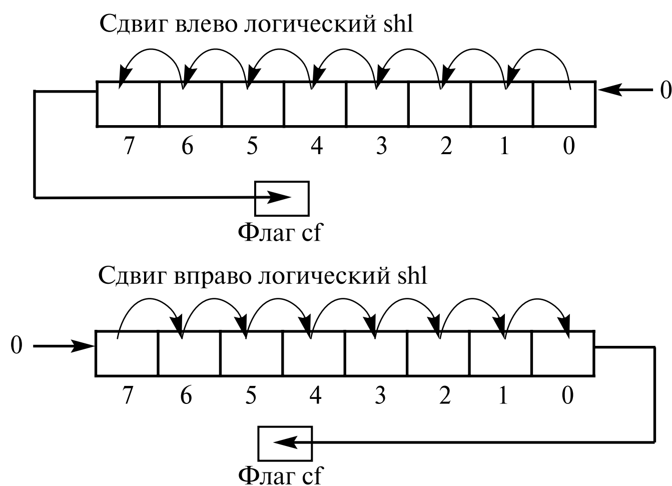


Рис. 30. Схема работы команд линейного логического сдвига

Команды арифметического линейного сдвига отличаются от команд логического сдвига тем, что они особым образом работаются со знаковым разрядом операнда.

1) **sal** операнд, счетчик_сдвигов (Shift Arithmetic Left) — арифметический сдвиг влево. Содержимое операнда сдвигается влево на количество битов, определяемое значением счетчик_сдвигов. Справа (в позицию младшего бита) вписываются нули. Команда sal **не сохраняет знака**, но *устанавливает флаг cf в случае смены знака* очередным выдвигаемым битом. В остальном команда sal полностью аналогична команде shl;
2) **sar** операнд, счетчик_сдвигов (Shift Arithmetic Right) — арифметический сдвиг вправо. Содержимое операнда сдвигается вправо на количество битов, определяемое значением *счетчик_сдвигов*. Слева в операнд вписываются нули. Команда sar **сохраняет знак**, восстанавливая его после сдвига каждого очередного бита.

На рисунке 31 показан принцип работы команд линейного арифметического сдвига.

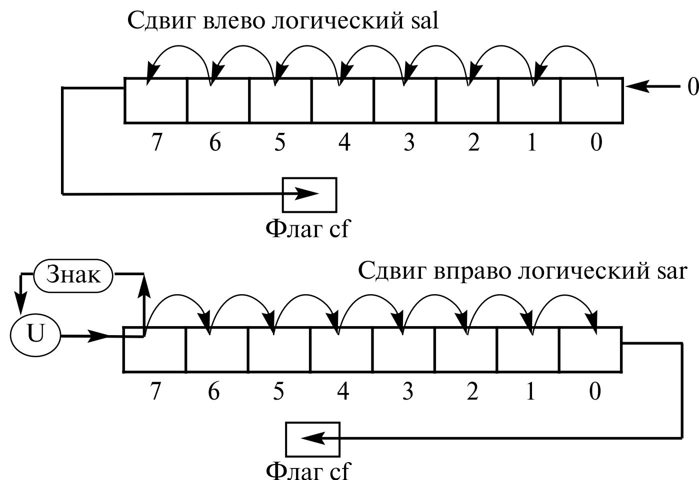


Рис. 31. Схема работы команд линейного арифметического сдвига

Команды циклического сдвига

К командам циклического сдвига относятся команды, сохраняющие значения сдвигаемых бит. Есть два типа команд циклического сдвига:

1) команды простого циклического сдвига;
2) команды циклического сдвига через флаг переноса cf.

К командам *простого циклического* сдвига относятся:

1) **rol** операнд,счетчик_сдвигов (Rotate Left) — циклический сдвиг влево. Содержимое операнда сдвигается влево на количество бит, определяемое операндом *счетчик_сдвигов*. Сдвигаемые влево биты записываются в тот же операнд справа;

2) **ror** операнд,счетчик_сдвигов (Rotate Right) — циклический сдвиг вправо. Содержимое операнда сдвигается вправо на количество бит, определяемое операндом *счетчик_сдвигов*. Сдвигаемые вправо биты записываются в тот же операнд слева.

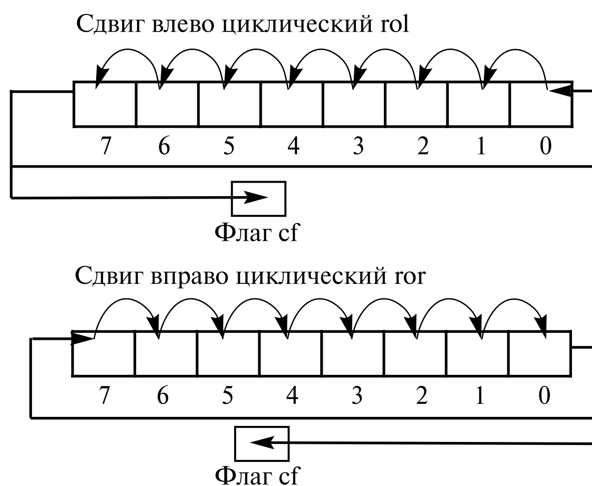


Рис. 32. Схема работы команд простого циклического сдвига

Как видно из рисунка 32, команды простого циклического сдвига в процессе своей работы осуществляют одно полезное действие, а именно: циклически сдвигаемый бит не только вдвигается в операнд с другого конца, но и одновременно его значение становится значением флага CF.

Команды циклического сдвига *через флаг переноса CF* отли чаются от команд простого циклического сдвига тем, что сдвигаемый бит не сразу попадает в операнд с другого его конца, а записывается сначала в флаг переноса CF. *Лишь следующее исполнение данной команды сдвига (при условии, что она выполняется в цикле) приводит к помещению выдвинутого ранее бита с другого конца операнда* (рис. 33).

К командам циклического сдвига *через флаг переноса cf* относятся следующие:

1) **rcl** операнд, счетчик_сдвигов (Rotate through Carry Left) — циклический сдвиг влево через перенос.

Содержимое операнда сдвигается влево на количество бит, определяемое операндом *счетчик_сдвигов*. Сдвигаемые биты поочередно становятся значением флага переноса cf.

2) **rcr** операнд, счетчик_сдвигов (Rotate through Carry Right) — циклический сдвиг вправо через перенос.

Содержимое операнда сдвигается вправо на количество бит, определяемое операндом *счетчик_сдвигов*. Сдвигаемые биты поочередно становятся значением флага переноса CF.

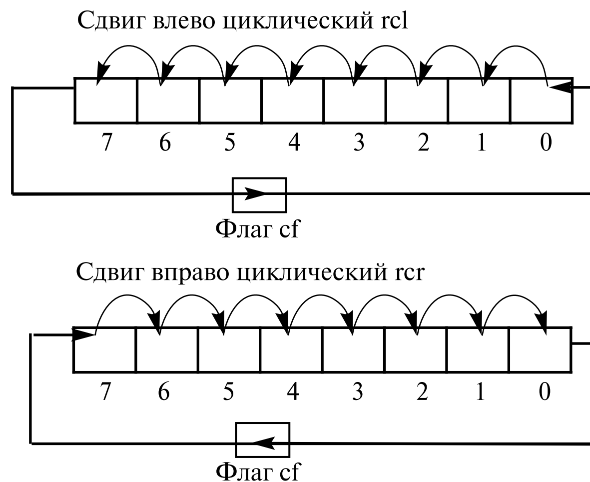


Рис. 33. Команды циклического сдвига через флаг переноса CF

Из рисунка 33 видно, что при сдвиге через флаг переноса появляется промежуточный элемент, с помощью которого, в частности, можно производить подмену циклически сдвигаемых битов, в частности, *рассогласование* битовых последовательностей.

Под рассогласованием битовой последовательности здесь и далее подразумевается действие, которое позволяет некоторым образом локализовать и извлечь нужные участки этой последовательности и записать их в другое место.

Дополнительные команды сдвига

Система команд последних моделей микропроцессоров Intel, начиная с i80386, содержит дополнительные команды сдвига, расширяющие возможности, рассмотренные нами ранее.

Это — команды сдвигов *двойной точности*:

1) **shld** операнд_1,операнд_2,счетчик_сдвигов — сдвиг влево двойной точности. Команда shld производит замену путем сдвига битов операнда операнд_1 влево, заполняя его биты справа значениями битов, вытесняемых из операнд_2 согласно схеме на рис. 34. Количество сдвигаемых бит определяется значением *счетчик_сдвигов*, которое может лежать в диапазоне **0 ... 31**. Это значение может задаваться непосредственным операндом или содержаться в регистре cl. Значение *операнд*_2 не изменяется.

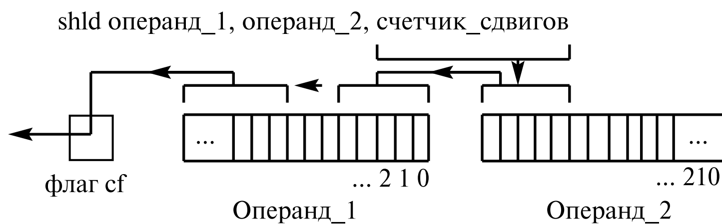


Рис. 34. Схема работы команды shld

2) **shrd** операнд_1,операнд_2,счетчик_сдвигов — сдвиг вправо двойной точности. Команда производит замену путем сдвига битов операнда операнд_1 вправо, заполняя его биты слева значениями битов, вытесняемых из операнд_2 согласно схеме на рисунке 35. Количество сдвигаемых бит определяется значением *счетчик_сдвигов*, которое может лежать в диапазоне **0 ... 31**. Это значение может задаваться непосредственным операндом или содержаться в регистре cl. Значение *операнд*_2 не изменяется.

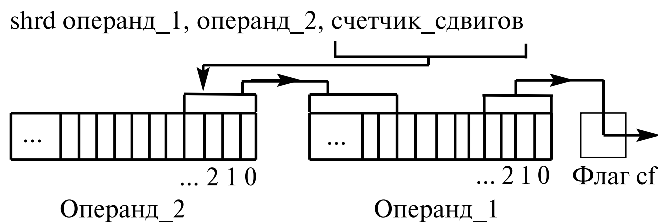


Рис. 35. Схема работы команды shrd

Как мы отметили, команды shld и shrd осуществляют сдвиги до 32 разрядов, но за счет особенностей задания операндов и алгоритма работы эти команды можно использовать для работы с полями длиной до 64 бит.

2. Команды передачи управления

Мы познакомились с некоторыми командами, из которых формируются *линейные* участки программы. Каждая из них в общем случае выполняет некоторые действия по преобразованию или пересылке данных, после чего микропроцессор передает управление следующей команде. Но очень мало программ работает таким последовательным образом. Обычно в программе есть точки, в которых нужно принять решение о том, какая команда будет выполняться следующей. Это решение может быть:

1) *безусловным* — в данной точке необходимо передать управление не той команде, которая идет следующей, а другой, которая находится на некотором удалении от текущей команды;
2) *условным* — решение о том, какая команда будет выполняться следующей, принимается на основе анализа некоторых условий или данных.

Программа представляет собой последовательность команд и данных, занимающих определенное пространство оперативной памяти. Это пространство памяти может быть либо непрерывным, либо состоять из нескольких фрагментов.

То, какая команда программы должна выполняться следующей, микропроцессор узнает по содержимому пары регистров *cs:(e)ip:*

1) cs — сегментный регистр кода, в котором находится физический (базовый) адрес текущего сегмента кода;
2) eip/ip — регистр указателя команды, в котором находится значение, представляющее собой смещение в памяти следующей команды, подлежащей выполнению, относительно начала текущего сегмента кода.

Какой конкретно регистр будет использоваться, зависит от установленного режима адресации use16 или use32. Если указано use16, то используется ip, если use32, то используется eip.

Таким образом, команды передачи управления изменяют содержимое регистров cs и eip/ip, в результате чего микропроцессор выбирает для выполнения не следующую по порядку команду про-

граммы, а команду в некотором другом участке программы. Конвейер внутри микропроцессора при этом сбрасывается.

По принципу действия команды микропроцессора, обеспечивающие организацию переходов в программе, можно разделить на 3 группы:

1. Команды безусловной передачи управления:
1) команда безусловного перехода;
2) команда вызова процедуры и возврата из процедуры;
3) команда вызова программных прерываний и возврата из программных прерываний.

2. Команды условной передачи управления:
1) команды перехода по результату команды сравнения cmp;
2) команды перехода по состоянию определенного флага;
3) команды перехода по содержимому регистра ecx/cx.

3. Команды управления циклом:
1) команда организации цикла со счетчиком ecx/cx;
2) команда организации цикла со счетчиком ecx/cx с возможностью досрочного выхода из цикла по дополнительному условию.

Безусловные переходы

Предыдущее обсуждение выявило некоторые детали механизма перехода. Команды перехода модифицируют регистр указателя команды eip/ip и, возможно, сегментный регистр кода **cs**. Что именно должно подвергнуться модификации, зависит:

1) от типа операнда в команде безусловного перехода (ближний или дальний);
2) от указания перед адресом перехода (в команде перехода) *модификатора*; при этом сам адрес перехода может находиться либо непосредственно в команде (прямой переход), либо в регистре или ячейке памяти (косвенный переход).

Модификатор может принимать следующие значения:

1) **near ptr** — прямой переход на метку внутри текущего сегмента кода. Модифицируется только регистр eip/ip (в зависимости от заданного типа сегмента кода use16 или use32) на основе указанного в команде адреса (метки) или выражения, использующего символ извлечения значения — $;
2) **far ptr** — прямой переход на метку в другом сегменте кода. Адрес перехода задается в виде непосредственного операнда

или адреса (метки) и состоит из 16-битного селектора и 16/32-битного смещения, которые загружаются, соответственно, в регистры cs и ip/eip;

3) **word ptr** — косвенный переход на метку внутри текущего сегмента кода. Модифицируется (значением смещения из памяти по указанному в команде адресу, или из регистра) только eip/ip. Размер смещения 16 или 32 бит;

4) **dword ptr** — косвенный переход на метку в другом сегменте кода. Модифицируются (значением из памяти — и только из памяти, из регистра нельзя) оба регистра — cs и eip/ip. Первое слово/двойное слово этого адреса представляет смещение и загружается в ip/eip; второе/третье слово загружается в cs.

Команда безусловного перехода jmp

Синтаксис команды безусловного перехода — **jmp [модификатор] адрес_перехода** — безусловный переход без сохранения информации о точке возврата.

Адрес_перехода представляет собой адрес в виде метки либо адрес области памяти, в которой находится указатель перехода.

Всего в системе команд микропроцессора есть несколько кодов машинных команд безусловного перехода **jmp**.

Их различия определяются дальностью перехода и способом задания целевого адреса. *Дальность* перехода определяется местоположением операнда *адрес_перехода*. Этот адрес может находиться в текущем сегменте кода или в некотором другом сегменте. В первом случае переход называется *внутрисегментным*, или *близким*, во втором — *межсегментным*, или *дальним*. Внутрисегментный переход предполагает, что изменяется только содержимое регистра eip/ip.

Можно выделить три варианта внутрисегментного использования команды jmp:

1) прямой короткий;
2) прямой;
3) косвенный.

Процедуры

В языке ассемблера есть несколько средств, решающих проблему дублирования участков программного кода. К ним относятся:

1) механизм процедур;
2) макроассемблер;
3) механизм прерываний.

Процедура, часто называемая также *подпрограммой*, — это основная функциональная единица декомпозиции (разделения на несколько частей) некоторой задачи. Процедура представляет собой группу команд для решения конкретной подзадачи и обладает средствами получения управления из точки вызова задачи более высокого уровня и возврата управления в эту точку.

В простейшем случае программа может состоять из одной процедуры. Другими словами, процедуру можно определить как правильным образом оформленную совокупность команд, которая, будучи однократно описана, при необходимости может быть вызвана в любом месте программы.

Для описания последовательности команд в виде процедуры в языке ассемблера используются две директивы: **PROC** и **ENDP**.

Синтаксис описания процедуры таков (рис. 36).

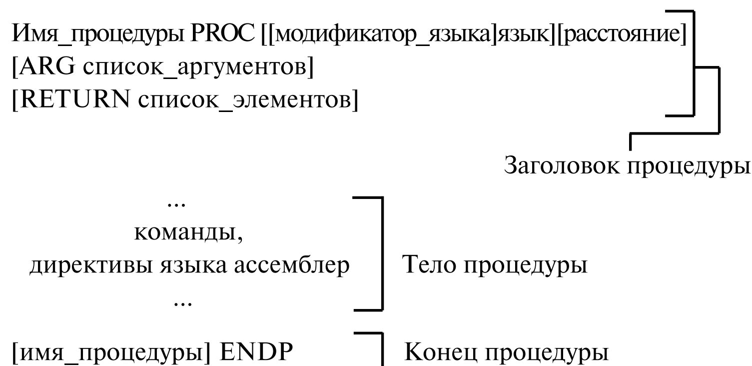


Рис. 36. Синтаксис описания процедуры в программе

Из рисунка 36 видно, что в заголовке процедуры (директиве PROC) обязательным является только задание имени процедуры. Среди большого количества операндов директивы PROC следует особо выделить **[расстояние]**. Этот атрибут может принимать значения **near** или **far** и характеризует возможность обращения к процедуре из другого сегмента кода. По умолчанию атрибут **[расстояние]** принимает значение **near**.

Процедура может размещаться в любом месте программы, но так, чтобы на нее случайным образом не попало управление. Если

процедуру просто вставить в общий поток команд, то микропроцессор будет воспринимать команды процедуры как часть этого потока и, соответственно, будет осуществлять выполнение команд процедуры.

Условные переходы

Микропроцессор имеет 18 команд условного перехода. Эти команды позволяют проверить:

1) отношение между операндами со знаком («больше — меньше»);
2) отношение между операндами без знака («выше — ниже»);
3) состояния арифметических флагов ZF, SF, CF, OF, PF (но не AF).

Команды условного перехода имеют одинаковый синтаксис:

jcc метка_перехода

Как видно, мнемокод всех команд начинается с «j» — от слова *jump* (прыжок), *cc* — определяет конкретное условие, анализируемое командой.

Что касается операнда *метка_перехода*, то эта метка может находиться только в пределах текущего сегмента кода, межсегментная передача управления в условных переходах не допускается. В связи с этим отпадает вопрос о модификаторе, который присутствовал в синтаксисе команд безусловного перехода. В ранних моделях микропроцессора (i8086, i80186 и i80286) команды условного перехода могли осуществлять только короткие переходы — на расстояние от —128 до +127 байт от команды, следующей за командой условного перехода. Начиная с модели микропроцессора 80386, это ограничение снято, но, как видите, только в пределах текущего сегмента кода.

Для того чтобы принять решение о том, куда будет передано управление командой условного перехода, предварительно должно быть сформировано условие, на основании которого и будет приниматься решение о передаче управления.

Источниками такого условия могут быть:

1) любая команда, изменяющая состояние арифметических флагов;

2) команда сравнения **cmp**, сравнивающая значения двух операндов;

3) состояние регистра ecx/cx.

Команда сравнения cmp

Команда сравнения **cmp** имеет интересный принцип работы. Он абсолютно такой же, как и у команды вычитания — **sub** операнд_1, операнд_2.

Команда **cmp** так же, как и команда **sub**, выполняет вычитание операндов и устанавливает флаги. Единственное, чего она не делает — это запись результата вычитания на место первого операнда.

Синтаксис команды cmp — **cmp операнд_1, операнд_2** (compare) — сравнивает два операнда и по результатам сравнения устанавливает флаги.

Флаги, устанавливаемые командой **cmp**, можно анализировать специальными командами условного перехода. Прежде чем мы их рассмотрим, уделим немного внимания мнемонике этих команд условного перехода (табл. 16). Понимание обозначений при формировании названия команд условного перехода (элемент в названии команды **jcc**, обозначенный нами **cc**) облегчит их запоминание и дальнейшее практическое использование.

Значение аббревиатур в названии команды jcc *Таблица 16*

Мнемоническое обозначение	**Английский**	**Русский**	**Тип операндов**
E e	equal	Равно	Любые
N n	not	Не	Любые
G g	greater	Больше	Числа со знаком
L l	less	Меньше	Числа со знаком
A a	above	Выше, в смысле «больше»	Числа без знака
B b	below	Ниже, в смысле «меньше»	Числа без знака

Перечень команд условного перехода
для команды cmp операнд_1, операнд_2

Таблица 17

<table>
<tr><th>Типы операндов</th><th>Мнемокод команды условного перехода</th><th>Критерий условного перехода</th><th>Значения флагов для осуществления перехода</th></tr>
<tr><td>Любые</td><td>je</td><td rowspan="2">операнд_1 = =операнд_2</td><td>zf = 1</td></tr>
<tr><td>Любые</td><td>jne</td><td rowspan="3">zf = 0</td></tr>
<tr><td>Со знаком</td><td>jl/jnge</td><td>операнд_1 < < операнд_2</td></tr>
<tr><td>Со знаком</td><td>jle/jng</td><td rowspan="3">операнд_1 <= <=операнд_2</td></tr>
<tr><td>Со знаком</td><td>jg/jnle</td><td>sf = of and zf = 0</td></tr>
<tr><td>Со знаком</td><td>jge/jnl</td><td>sf = of</td></tr>
<tr><td>Без знака</td><td>jb/jnae</td><td>операнд_1 < < операнд_2</td><td>cf = 1</td></tr>
<tr><td>Без знака</td><td>jbe/jna</td><td rowspan="3">операнд_1 <= <=операнд_2</td><td>cf = 1 or zf=1</td></tr>
<tr><td>Без знака</td><td>ja/jnbe</td><td>cf = 0 and zf = 0</td></tr>
<tr><td>Без знака</td><td>jae/jnb</td><td>cf = 0</td></tr>
</table>

Не удивляйтесь тому обстоятельству, что одинаковым значениям флагов соответствует несколько разных мнемокодов команд условного перехода (они отделены друг от друга косой чертой в табл. 17). Разница в названии обусловлена желанием разработчиков микропроцессора облегчить использование команд условного перехода в сочетании с определенными группами команд. Поэтому разные названия отражают скорее различную функциональную направленность. Тем не менее то, что эти команды реагируют на одни и те же флаги, делает их абсолютно эквивалентными и равноправными в программе. Поэтому в таблице 17 они сгруппированы не по названиям, а по значениям флагов (условиям), на которые они реагируют.

Команды условного перехода и флаги

Мнемоническое обозначение некоторых команд условного перехода отражает название флага, с которым они работают, и имеет

следующую структуру: первым идет символ «**j**» (*jump*, переход), вторым — либо обозначение флага, либо символ отрицания «**n**», после которого стоит название флага. Такая структура команды отражает ее назначение. Если символа «n» нет, то проверяется состояние флага, если он равен 1, производится переход на метку перехода. Если символ «n» присутствует, то проверяется состояние флага на равенство 0, и в случае успеха производится переход на метку перехода.

Мнемокоды команд, названия флагов и условия переходов приведены в таблице 18. Эти команды можно использовать после любых команд, изменяющих указанные флаги.

Таблица 18

Команды условного перехода и флаги

Название флага	**Номер бита в eflags/flags**	**Команда условного перехода**	**Значение флага для осуществления перехода**
Флаг переноса cf	1	jc	cf = 1
Флаг четности pf	2	jp	pf = 1
Флаг нуля zf	6	jz	zf = 1
Флаг знака sf	7	js	sf = 1
Флаг переполнения of	11	jo	of = 1
Флаг переноса cf	1	jnc	cf = 0
Флаг четности pf	2	jnp	pf = 0
Флаг нуля zf	6	jnz	zf = 0
Флаг знака sf	7	jns	sf = 0
Флаг переполнения of	11	jno	of = 0

Если внимательно посмотреть на таблицы 17 и 18, видно, что многие команды условного перехода в них являются эквивалентными, так как в основе и тех, и других лежит анализ одинаковых флагов.

Команды условного перехода и регистр ecx/cx

Архитектура микропроцессора предполагает специфическое использование многих регистров. К примеру, регистр **EAX/AX/AL** используется как аккумулятор, а регистры **BP, SP** — для работы со стеком. Регистр **ECX/CX** тоже имеет определенное функциональное назначение: он выполняет роль *счетчика* в командах

управления циклами и при работе с цепочками символов. Возможно, что функционально команду условного перехода, связанную с регистром **ecx/cx**, правильнее было бы отнести к этой группе команд.

Синтаксис этой команды условного перехода таков:

1) **jcxz** метка_перехода (Jump if cx is Zero) — переход, если cx нуль;

2) **jecxz** метка_перехода (Jump Equal ecx Zero) — переход, если **ecx** нуль.

Эти команды очень удобно использовать при организации цикла и при работе с цепочками символов.

Нужно отметить ограничение, свойственное команде **jcxz/jecxz**. В отличие от других команд условной передачи управления команда **jcxz/jecxz** может адресовать только короткие переходы — на –128 байт или на +127 байт от следующей за ней команды.

Организация циклов

Цикл, как известно, представляет собой важную алгоритмическую структуру, без использования которой не обходится, наверное, ни одна программа. Организовать циклическое выполнение некоторого участка программы можно, к примеру, используя команды условной передачи управления или команду безусловного перехода **jmp**. При такой организации цикла все операции по его организации выполняются вручную. Но, учитывая важность такого алгоритмического элемента, как цикл, разработчики микропроцессора ввели в систему команд группу из трех команд, облегчающую программирование циклов. Эти команды также используют регистр **ecx/cx** как *счетчик цикла*.

Дадим краткую характеристику этим командам:

1) **loop** метка_перехода (Loop) — повторить цикл. Команда позволяет организовать циклы, подобные циклам for в языках высокого уровня с автоматическим уменьшением счетчика цикла. Работа команды заключается в выполнении следующих действий:

   а) декремента регистра ECX/CX;

   б) сравнения регистра ECX/CX с нулем: если (ECX/CX) = = 0, то управление передается на следующую после loop команду;

2) **loope/loopz** метка_перехода

Команды loope и loopz — абсолютные синонимы. Работа команд заключается в выполнении следующих действий:

а) декремента регистра ECX/CX;

б) сравнения регистра ECX/CX с нулем;

в) анализа состояния флага нуля ZF если (ECX/CX) = 0 или XF= 0, управление передается на следующую после loop команду.

3) **loopne/loopnz** метка_перехода

Команды loopne и loopnz также абсолютные синонимы. Работа команд заключается в выполнении следующих действий:

а) декремента регистра ECX/CX;

б) сравнения регистра ECX/CX с нулем;

в) анализа состояния флага нуля ZF: если (ECX/CX) = 0 или ZF=1, управление передается на следующую после loop команду.

Команды **loope/loopz** и **loopne/loopnz** по принципу своей работы являются взаимообратными. Они расширяют действие команды **loop** тем, что дополнительно анализируют флаг zf, что дает возможность организовать досрочный выход из цикла, используя этот флаг в качестве индикатора.

Недостаток команд организации цикла **loop**, **loope/loopz** и **loopne/loopnz** состоит в том, что они реализуют только короткие переходы (от —128 до +127 байт). Для работы с длинными циклами придется использовать команды условного перехода и команду jmp, поэтому постарайтесь освоить оба способа организации циклов.

Содержание

ЛЕКЦИЯ № 1. Введение в информатику 3
1. Информатика. Информация.
Представление и обработка информации 3
2. Системы счисления 4
3. Представление чисел в ЭВМ 5
4. Формализованное понятие алгоритма 5

ЛЕКЦИЯ № 2. Язык Pascal 7
1. Введение в язык Pascal 7
2. Стандартные процедуры и функции 10
3. Операторы языка Pascal. 12

ЛЕКЦИЯ № 3. Процедуры и функции 16
1. Понятие вспомогательного алгоритма 16
2. Процедуры в Pascal 17
3. Функции в Pascal 17
4. Опережающие описания
и подключение подпрограмм. Директива 18

ЛЕКЦИЯ № 4. Подпрограммы 20
1. Параметры подпрограмм 20
2. Типы параметров подпрограмм 20

ЛЕКЦИЯ № 5. Строковый тип данных 24
1. Строковый тип в Pascal 24

2. Процедуры и функции
для переменных строкового типа 24
3. Записи 25
4. Множества 26

ЛЕКЦИЯ № 6. Файлы 29
1. Файлы. Операции с файлами 29
2. Модули. Виды модулей 34

ЛЕКЦИЯ № 7. Динамическая память **. 38**
1. Ссылочный тип данных.
Динамическая память.
Динамические переменные 38
2. Работа с динамической памятью
Нетипизированные указатели. 39

ЛЕКЦИЯ № 8. Абстрактные структуры данных **. . . . . . .** 40
1. Абстрактные структуры данных 40
2. Стеки 41
3. Очереди 42
ЛЕКЦИЯ № 9. Древовидные структуры данных **. . . . . .** 45
1. Древовидные структуры данных 45
2. Операции над деревьями 46
3. Примеры реализации операций 47

ЛЕКЦИЯ № 10. Графы**. . . . .** 51
1. Понятие графа.
Способы представления графа в ЭВМ 51

2. Представление графа
списком инцидентности.
Алгоритм обхода графа в глубину 53
3. Представление графа списком списков.
Алгоритм обхода графа в ширину 54

ЛЕКЦИЯ № 11. Объектный тип данных 56
1. Объектный тип в Pascal.
Понятие объекта, его описание
и использование 56
2. Наследование 59
3. Создание экземпляров объектов 60
4. Компоненты и область действия 62

ЛЕКЦИЯ № 12. Методы 64
1. Методы 64
2. Конструкторы и деструкторы 66
3. Деструкторы 68
4. Виртуальные методы 71
5. Поля данных объекта
и формальные параметры метода 73
ЛЕКЦИЯ № 13. Совместимость типов объектов 75
1. Инкапсуляция 75
2. Расширяющиеся объекты .76
3. Совместимость типов объектов 81

ЛЕКЦИЯ № 14. Ассемблер 84
1. Об ассемблере 84
2. Программная модель микропроцессора 85

3. Пользовательские регистры 86
4. Регистры общего назначения 88
5. Сегментные регистры 89
6. Регистры состояния и управления 90

ЛЕКЦИЯ № 15. Регистры 96
1. Системные регистры микропроцессора 96
2. Регистры управления 96
3. Регистры системных адресов 97
4. Регистры отладки 98

ЛЕКЦИЯ № 16. Программы на Ассемблере 100
1. Структура программы на ассемблере 100
2. Синтаксис ассемблера 100
3. Директивы сегментации .111

ЛЕКЦИЯ № 17. Структуры команд
на Ассемблере 121
1. Структура машинной команды 121
2. Способы задания операндов команды 126
3. Способы адресации 128

ЛЕКЦИЯ № 18. Команды 132
1. Команды пересылки данных 132
2. Арифметические команды 141

ЛЕКЦИЯ № 19. Команды передачи управления 165
1. Логические команды 165
2. Команды передачи управления 177