**Assembler**

**Вступ.**

Після створення процесора 8086 фірма Intel розробила більш досконалі процесори об’єднані під назвою I 80x86, така назва означає, що всі команди мікропроцесора, які виконуються на молодших моделях обов’язково виконуються на старше, отже все ПЗ, які розроблені для процесора 8086 успішно будуть працювати і на останніх моделях 80486 і Pentium. Ми будемо розглядати процесори з точки зору програміста. Не дивлячись на різноманітність моделей процесорів, найбільш важливим з точки зору біології програмування є 8086 як базова модель і 80386 , як перший процесор фірми Intel, який в повному об’ємі реалізував принцип багатозадачності.

## Програмування на мові асемблера

Програмування на мові асемблера вважається складною задачею, причини цього такі:

1. Мова асемблера будь-якого процесора суттєво складніша будь-якої мови високого рівня. Щоб скористатись всіма можливостями мови асемблера, треба по крайній мірі знати команди мікропроцесора, а їх число з усіма можливими варіантами переважає 100, їх кількість значно перевищує кількість операторів і ключових слів інших мов високого рівня. Проблема ускладнюється ще тим, що зміни в асемблері виникають набагато швидше ніж в мовах високого рівня, це зв’язано з появою нових мікропроцесорів і відповідно нових команд.
2. Програміст, який використовує мови асемблера повинен сам слідкувати за розподілом пам’яті та вмістом регістрів, щоб коректно розподіляти і оперувати пам’яттю, в мовах високого рівня це робиться автоматично при допомозі компілятора, але ця обставина має перевагу: можна оптимально розташувати дані в пам’яті, забезпечити максимальну швидкість виконання та мінімальну довжину програми.
3. Програми на мові асемблера важче проектувати та підлагоджувати, треба весь час пам’ятати, що конкретно знаходиться в кожному регістрів в даній комірці пам’яті. Прийнято вважати, що розробка програми тільки на мові асемблера, деякого процесора, навіть якщо він поширений не рекомендується. Зрозуміло, що будь-яку програму можна написати тільки з допомогою асемблера, але для цього треба використати набагато більшу кількість команд і час який піде на її виконання і відладку буде набагато більший ніж для мови високого рівня. Набагато вигідніше писати програми на мові високого рівня, а найбільш критичні частини на швидкодії писати на мові асемблера. Наприклад на асемблері можна скласти процедури для реалізації вводу-виводу низького рівня, процедури обробки переривань та деякі інші.

**Етапи створення програми.**

Розробка програми на мові асемблера включає кілька етапів.

1. Підготовка початкового тексту програми;
2. Асемблювання програми(отримання об’єктного коду);
3. Компоновка програми(отримання виконуваного файлу);
4. Відладка програми(знаходження помилок).

Ці етапи циклічно повторюються.

**Підготовка лістінгу програми.**

Текст програми на мові асемблера записується в один або кілька файлів, імена файлів і їх розширення можуть бути будь-які, прийнято для файлів які містять програми мовою асемблера прийнято використовувати розширення \*.asm. Для файлів визначених констант і повних типів розширення \*.inc. Ці файли є текстовими їх можна набрати в будь-якому текстовому редакторі. Можна також використати інтегровані середовища для розробки програм, при програмуванні зручно виділяти один каталог для збереження всіх файлів програми і звідти запускати всі необхідні програми для підготовки, асемблювання та компонування програми. При використанні стандартних редакторів тексту, редаговані тексти треба зберігати у вигляді звичайних файлів у форматі ASCII, це означає, що ці файли треба зберігати без додаткових символів, ці символи часто вставляють в текст різні спеціалізовані редактори, наприклад WORD.

**Асемблювання програми.**

Підготовлений текст мови асемблер є початковим для спеціальних програм, які називаються асемблерами, далі **програма асемблера**. Задача програми асемблера перетворити текст програми у форму двійкових команд, останні можуть вже бути виконанні мікропроцесором. Після асемблювання дістають так звані файли об’єктних модулів, які мають відповідні розширення \*.obj. Для отримання об’єктних файлів необхідного виконати відповідну програму асемблера masm фірми Microsoft, або tasm фірми Borland. В обох випадках після команди вказується ім’я файлу : **masm prog1.asm**, **tasm prog1.asm**.

Така форма команди є мінімальною, крім цієї форми можна використовувати іншу форму задаючи перед іменем файлу опції або ключі. Якщо програма складається з декількох файлів, то їх асемблювання проводиться незалежно один від одного, хоча отримані об’єктні файли представлені вже в двійковій формі запускати їх на виконання не можливо.

**Компоновка програми.**

Текст програми може знаходитись в декількох файлах, змінні які описані в цих файлах можуть використовуватися спільно, якщо такі файли асемблюються окремо, то не можливо дістати повну інформацію, для того, щоб генерувати виконавчий код. Тому процес підготовки програми обов’язково включає в себе етап компоновки. На цьому етапі визначають всі невідомі, при окремому асемблювані, адреси всіх змінних або функцій, які використовуються спільно. Процес об’єднання об’єктних модулів в один файл виконується спец. програмою, яка називається компоновщиком. Це може бути програма link фірми Microsoft, або tlink фірми Borland, отримуваний виконуваний файл має розширення \*.exe, або \*.com. Компановщику треба передати імена відповідних об’єктних файлів.

### Link prog1.obj prog2.obj

**Tlink prog1.obj prog2.obj**

Після компонування отримується виконуваний файл, він отримує ім’я файла, який стоїть перший у відповідній команді.**Відладка програми.**

За винятком початкових простих програм практично всі програми на мові асемблера мають потребу в відладці. Для відладки можна використовувати різні відладчики, наприклад **tg386-Turbo Debuger** фірми Borland. Сучасні відладчики дозволяють в процесі відладки контролювати значення регістрів загального призначення, а також значення змінних і змінювати їх в процесі відладки, можна переглядати зміст різних ділянок пам’яті, можна контролювати виконання покроково, або розставляти точки зупинки.**Використання інтегрованих середовищ (ІС).**

Дуже зручно користуватись для підготовки тексту програми редакторами ІС, такі можливості передбачають практично всі виробники сучасних асемблерів. ІС мають суттєву перевагу, так як дають доступ до довідкової інформації. Вони дозволяють також зразу асемблювати та компонувати набрані тексти і провести його відладку. Потім знову можна повернутись до його редагування, при цьому складається ілюзія роботи з однією програмою. Слід відмітити, що відладчики вбудовані в ІС мають дещо обмежені можливості.

**Основні відомості та правила для написання програм на асемблері.**

Всі данні в мікропроцесорі представляються у вигляді набору бітів певної довжини, тільки окремі команди мікропроцесора розглядають вміст регістру або комірки пам’яті в певному форматі. В переважній більшості випадків сам програміст вирішує як розглядати певний набір бітів: як код клавіш, як число без знаку чи зі знаком, або як двійково-десяткове число. Всі числа які представляють певні данні розглядаються в позиційній системі числення. При представленні чисел команді мікропроцесора використовується двійкова система числення. Щоб спростити процес програмування мова асемблера дає можливість данні записувати так, як це зручно програмісту, при цьому можна записувати данні в різних системах числення. Можна навіть замість числа записувати вирази, процес перетворення цих чисел у двійковий вигляд буде виконувати програма асемблера. При записі двійкових чисел використовують суфікс **В** або ***в*: 11011111В, 11010101*в.***

При записі чисел в 16-вій системі числення треба враховувати, що для чисел 10-15ами.За винятком пбукви латинського алфавіту. Щоб відрізнити 16-ві числа дописується суфікс **Н** або **h,** крім того перш0пбуква в записі шістнадцяткового числа повинн0пбути арабською цифрою. Тобто для написання числа c5h не достатньо суфікса h, треба обов’язково поставити спершу 0:0C5h. За умовчанням використовується десяткове CЧ. При допомозі директиви **radix** можна міняти CЧ.Двійкові числа.Для великої кількості команд допускається, що інформація представлена в двійкових кодах, довжина цих двійкових чисел може бути різною. Мікропроцесори сімейства І 80х86 підтримують дані таких розмірностей:

1. Байти;
2. Слова;
3. Подвійні слова;
4. Рядки.

Крім того до специфічних даних можна віднести адрес переходів. Байт – це мінімальна к-сть інформації при роботі мікропроцесора, складається із 8 біт. Вся пам’ять мікропроцесора може розглядатись як велика к-сть байтів.

210б=1Кб 220б=1Мб 230б=1Гб 240б=1Тб

Так як байт складається з 8 біт, кожен біт може перебувати в двох станах, то байт може перебувати в 256-тьох станах. Крім представлення чисел байт використовується для збереження символьних кодів, наприклад це може бути символ, що набирається з клавіатури. 256 різних значень байта дозволяють описати з допомогою ASCII не тільки десяткові числа, а й весь англійський алфавіт, а також знаки пунктуації, спеціальні знаки, управляючі символи, символи національних алфавітів. Слово може містити 216=65536 різних станів, а це є 64 Кб. Слово це найбільш зручний спосіб збереження інформації в даному сімействі мікропроцесорів, адже їх внутрішні регістри як мінімум 16-ти розрядні. У всіх мікропроцесорів фірми Intel правило запису багатобайтних чисел таке: *молодший байт знаходиться за молодшою адресою в пам’яті, старший за старшою адресою.* Адреса деякої ділянки пам’яті яка розглядається як слова рівна адресі молодшого байту. Наприклад, якщо за адресою N записано слово 1539H, то це означає, що за адресою N записаний байт 39H, а за адресою N+1 – 15Н. Важливим є число 65536, для процесорів 80х86 це число визначає максимальну довжину сегменту, тобто області пам’яті, яка містить код або данні програми, тому за допомогою таких чисел до 65536 можна адресувати комірки пам’яті всередині даного сегменту, така адресація, коли використовується зміщення всередині сегменту називається **ближньою – Near**. Подвійні слова рівні 4 байт, або 32 біт. Одне слово може мати 4294967296=4Мб. Подвійні слова зберігаються в пам’яті за цим же правилом: *молодший байт за молодшою адресою, байти в пам’яті розташинятком ппослідовно один за одним* 12554959Н – слово подвійне, якщо воно знаходиться за адресою N, то за адресою N знаходиться байт 59Н, N+1 – 49H, N+2 – 55H, N+3 – 12H. Подвійне слово при програмуванні для процесорів молодших 80386 має достатньо місця для збереження не тільки адреси всередині сегменту, а й адреси початку сегменту і адреси зміщення в середині сегменту називається**дальньою – for adress** адресою. При програмуванні , коли для завдання адреси використовується 32 біти (для мікропроцесорів 80386 і старше) в подвійному слові кожна зберігати 32-розрядне зміщення. За допомогою цього зміщення можна звертатись до будь-якої комірки пам’яті. Рядки в мікропроцесорах родини І 80х86 можна виконувати операції над рядками. Рядок – це послідовність байтів або слів, а для мікропроцесорів старших 80386 подвійних слів, які знаходяться в пам’яті комп’ткера. Мінімальна довжина рядку 1 елемент, максимальна довжина рядку може досягати 64 Кб для мікропроцесорів молодших 80386, і 4Гб для мікропроцесорів старших 80386.

**Представлення чисел зі знаком.**

Мікропроцесори даної родини підтримають обробку чисел зі знаком, для чисел, які мають розмірність байт, слово, подвійне слово. Для представлення знаку використовують самий старший розряд числа 0 – “+”, 1 – “-“. В зв’язку з цим діапазон представлення чисел звужується. Наприклад для байтів зі знаком мінім. число : -128, макс. число : 127. Відповідно слово зі знаком : -32768 і 32768, подвійне слово : -2147483648 і 2147483647.

Від’ємні числа зберігаються в додатковому коді. Використання двійквого коду має переваги: можна виконувати арифметичні операції не знаючи як представлені операнди: зі знаком чи без знаку. Це означає, що сам програміст вирішує як інтерпретувати результат зі знаком чи без. **Двійково-десяткові числа.** Мікропроцесори даної родини дозволяють працювати з так званими двійково-десятковими числами, вони представляють собою числа від 0 до 9 записані в двійковому вигляді, для їх запису використовується 4 біти. На відміну від просто двійкових чисел тут не ми.За винятком птетради, які відповідають числам a – f 16-вої СЧ. Мікропроцесори цьої родини підтримують роботу з двома форматами двійково-десяткових чисел: унакованими і неунакованими двійково-десятковими числами.

Вони займають 1б – 2птетради.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

0 0

отже при допомозі однобайтного двійково-десяткового числа можна записати десяткові числа 0 – 99.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 |

9 9

При необхідності працювати з більшими числами необхідно використовувати кілька байт. Зауважимо, що не має стандартної форми запису для від’ємного представлення таких чисел: старший біт тут не можна використовувати як знаковий. Програміст сам повинен турбуватися про збереження інформації про знак числа і самостійно обробляти знак при виконанні арифметичних дій. МП для такої обробки не мають відповідних команд.

**Неунаковані двійково-десяткові числа.**

Як і унаковані вони займають 1б, але в цьому байті записується тільки одна десяткова цифра, ця цифра записується в молодшій тетраді:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |

0 – 9

Отже діапазон представлення чисел 0 – 9. Значення старшоWптетради не суттєве:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| х | х | х | Х | 1 | 0 | 0 | 1 |

9

Для зручності відладки в старшCптетраду записують 0:

0 0 0 0

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Х | х | х | Х | 1 | 0 | 0 | 1 |

0 –9

Зворотне представлення просте: молодш0птетрада, якщо вона знаходиться у межах представлення чисел 0 – 9 визначає відповідне двійково-десяткове неунаковане число. При необхідності працювати з великою розрядністю таких чисел необхідно використовувати також кілька байт.

**Лекція за 24.09.98.**

**Приклад структури програми на мові асемблера (ш0блон).**

Приведемо основні директиви, які використовуються і програмі на мові асемблера. Тепер використовуються спрощені способи опису сегментів. Сегменти – це частини програми (сегменти даних). Текст програми можна набирати великими або малими буквами, програма асемблера самостійно переводить текст у верхній регістр. Обов’язково у тексті програми писати коментарі. Текст який стоїть за символом “;” є коментарем, він ігнорується при асемблюванні програми. Директива яка як правило йде з початку програми **dosseg** описує розташування сегментів. Директива **mode small** – директива опису моделі пам’яті, наприклад в цьому випадку говорять, що модель пам’яті типу **small**, тобто є один сегмент коду і один сегмент коду і один сегмент даних. Директива **.stack 100** – директива, яка задає початок сегменту стеку. В стеці зберігаються тимчасові змінні і адреси повернення з підпрограми або переривань. Як правило будь-яка програма має сегмент стеку. В даному випадку такий запис директиви **.stack 100** позначає, що стек має довжину 100байтів. Точну величину стеку визначає число викликів підпрограм та системних ф-цій, стек треба збільшувати. В кінцевому підсумку розмір стеку підбирають експериментально, коли програма вже написана і відлагоджена. Директива **.data** – директива початку сегменту ініціалізованих даних, після цього рядка поміщають змінні значення яких відоме (ініціалізовані змінні). Директива **.data** **?** – директива початку сегменту неініціалізованих даних. Директива **.code** – початок сегменту коду. Сегмент коду – це частина коду яка містить команди МП, власне це й є сама програма. В деяких випадках в цей сегмент можна поміщати не тільки команди, а й дані. Наприклад у підпрограмах обробки переривань. Директива **.startup** – точка початку виконання програми. З наступного після даного рядка починається виконання програми після її завантаження в пам’ять, ця директива повинн0пбути обов’язково всередині сегмента коду, але не обов’язково н0пйого початку. Директива **.exit 0.** – це директива закінчення програми при виконанні цього рядка управління буде передане ОС. Таких директив в програмі може бути кілька, число після директиви може приймати кілька значень, це число після закінчення програми присвоюється змінній ОС з назвою **ERRORLEVEL**, ця змінна використовується в командних файлах. Є спеціальні команди для роботи з цією змінною. Використовуючи різні числа можна повідомляти ОС, яка причина закінчення програми. Директиви початку і закінчення програми є по суті макровизначеними, вони визначають необхідний код який автоматично додається на початок і кінець програми. Цей код можна подивитись у файлі лістингу після асемблювання. **End** – директива закінчення тексту програми на мові асемблера. Рядки які розміщені після директиви **end** можна писати коментар в довільній формі.

**Команди мови асемблера.**

Мінімальна одиниця тексту програми на мові асемблера – це рядок. Асемблер проглядає програму порядкові зверху в низ, кожний рядок закінчується символом переведення рядка Vпйого довжина в більшості випадків не може перевищувати 255 символів, однак зручно обмежувати довжину рядка широкою екрану (80 символів). При необхідності вводити в програмі довгі рядки їх можна об’єднувати, в кінець рядка при цьому потрібно включати символ **\**.

Рядок 1 \

Рядок 2 Рядок 1 Рядок 2.

В текст програми можна вставляти порожні рядки, які складаються тільки із розділювачів. В мові асемблера розділювачами служать символи пропуску та горизонтальна табуляція. При допомозі пустих рядків зручно розділяти логічні частини програми. Рядок може складатись тільки із коментарів, в такому випадку перший символ рядка обов’язково **;**. В кожному рядку може розміщуватись або команда мікропроцесора, яка записана у мнемонічному (словом) вигляді, або директива, директиви інакше називають псевдооператорами. Команда мікропроцесора – це інструкція яку мікропроцесор буде виконувати в процесі виконання програми. Асемблер переводить рядки з мнемоніками (буквеними записами команд) у послідовності байт, які безпосередньо може виконувати процесор. Директиви служать для визначення даних, які використовуються в програмі і для управління процесом асемблювання. Рядок у програмі на мові асемблера може складатися з 4 полів:

1. Поле мітки (поле імені);
2. Поле оператора (або псевдооператора);
3. Поле операндів;
4. Поле коментарів.

Обов’язковим є тільки поле оператора або псевдооператора, всі решту полів можуть бути відсутні. Мітка є спрощеною мнемонічною формою запису адреси команди в сегменті коду. Так саме ім’я є спрощеною формою запису вмісту деякої переважно в сегменті даних. Отже рядки в програмі записуються в такому загальному вигляді:

[< мітка >]: < команда > < операнди >; коментарі

< ім’я > <директива < операнди >; коментарі

Потрібно звернути увагу на відсутність “**:**” після імені. Поля в рядках мови асемблера розділяються між собою на крайній мірі одним пропуском. Розглянемо поля детальніше. **Поле мітки.** При визначенні в програмі мітки або імені змінної можна використовувати букви латинського алфавіту великі або малі, а також символи :? \_ @ $. Максимальна к-сть символів з яких може складатись мітка або ім’я 255, але відрізняються мітки та імена по перших 32символах. Доцільно в програмі використовувати осмислені імена. Зручно і програмі для швидкого розпізнавання міток від імені починати всі мітки з букви L і записувати їх тільки великими буквами. При записі імен змінних зручно використовувати префікс, який означає тип змінної, наприклад : **skp\_byte**. При програмуванні з використанням асемблера **masm**, або **tasm** можна використовувати будь-яке число локальних міток @@. Для переходу до таких міток треба використовувати зарезервовані слова: @F – forward; @B – backward. Перш0п@F – це перехід до наступної мітки, @B – це перехід до попередньої мітки. Мітка або ім’я змінної не повиннVпбути зарезервованим словом.

**Поле операндів.**

Поле операндів залежить від попереднього поля.

**Поле коментарів.**

**Структура програми на мові asm.**

При роботі будь-якої програми в будь-який момент часу можна виділяти декілька ділянок пам’яті з якими працює процесор. Використання сегментів є особливістю процесорів х86. Воно переслідує таку мету:ами.За вання сегментів є спробою заха вити незв’язані ділянки пам’яті в програмі. Допускалось, що для того, щоб визначити адресу деякої комірки пам’яті потрібно знати дві складові цієї адреси: адресу початкового сегменту та адресу зміщення комірки пам’яті від початку сегменту. Адреси початків сегментів знаходяться у відповідних сегментах регістрах CS, DS, SS, ES – змінюючи вміст цих регістрів можна дістати доступ до тих чи інших ділянок пам’яті, тому такий запис легко зламати. Тому в процесорі 386 в захащеному режимі роботи адреси початків сегментів, їх довжини та права доступу зберігаються в спеціальних системних одиницях. В сегментних регістрах зберігають тільки індекс структури всередині таблиці. Це дозволило ізолювати сегменти та обмежило доступ до них з програми. Адже для доступу потрібно знати не тільки вміст сегменту регістру, але й тієї структури на яку він вказує. Така зміна можлива тільки при використанні привілейованих команд, що виконується тільки в певних умовах, можуть виконуватись в ОС, але не можуть використовуватись в програмах користувача. Паралельно з цим збільшення розміру сегментів:амід 64Кб для 86, до 4Гб для 486. Крім того програма буде виконуватись, якщо змінити адресу початку сегменту. Це означає, що в пам’яті він буде перенесений в інше місце. При цьому відносне зміщення в сегменті не можна зміщувати. Це дозволить переносити програму в пам’яті навіть під час її виконання. Використання сегментації пам’яті дозволяє різко збільшити розміри пам’яті, яку можна використати. Так, в захащеному режимі максимально можливий об’єм пам’яті, яку можна адресувати – 64Тб. Це віртуальна пам’ять. Тоді як фізичний об’єм пам’яті складає 4Гб. В деяких випадках використовувати сегментацію недоцільно. Тому в старших моделях можна працювати із сегментами довжиною до 4Гб. Це означає, що вся фізична пам’ять розглядається як один сегмент. Повністю усі переваги сегментації пам’яті проявляється в захащеному режимі роботи. В реальному режимі роботи оперує з 4-ма сегментами:а

1. Сегмент коду;
2. Сегмент даних;
3. Сегмент стеку;
4. Сегмент додаткових даних.

Для процесорів 386 з’явились ще 2 сегменти додаткових даних. Ці сегменти в пам’яті можуть розташиняватись як завгодно по відношенню один до одного. Ніяких обмежень на їх розташування немає. Сегменти можуть починатись з однієї адреси, тобто можуть перекриватись. Вони можуть розташиняватись в пам’яті без перекривання, або між ними можуть бути проміжки.

Для визначення адреси наступної команди після виконання попередньої використовують 2 регістри: CS (регістр сегменту коду, він містить початок адресу цього сегменту), і IP(регістр вказівника команд, якість зміщення команди відносно початку сегменту). В процесі виконання програми вміст ІР автоматично змінюється. Якщо змінити вміст регістрів CS, та ІР, то виклик програми почнеться з іншої адреси. Але явним чином змінити вміст цих регістрів неможливо. Це можна зробити тільки при використані деяких команд. Наприклад при виклику підпрограми. Або виконані програми безумовного переходу. Як уже відмічалось в сегменті коду можна також описувати дані. Але це роблять тільки в обгрунтованих випадках, наприклад в програмі обробки переривань.

### Початок програми.

В початковій версії необхідно було відмітити точку входу в програму деякою міткою. Ім’я цієї мітки потім вказувалось в директиві **end**. Це було ознакою закінчення файлу з програмою. У сучасних асемблерах для визначення точки входу в програму використовується директива **startup**. Вона записується в сегменті коду, це спрощує запис і, крім цього, автоматично ініціалізує значення сегментних регістрів. В старих версіях це потрібно було робити вручну.

### Кінець програми.

Для закінчення програми потрібно було скористяватись стандартною директивою. Вона робить відповідний виклик автоматично. Якщо програму необхідно закінчити спеціальним чином, наприклад встановити резидентну програму, то користяватись цією директивою не можна. В таких випадках потрібно явно викликати відповідну функцію операційної системи.

### Змінні в програмі.

В усіх інших сегментах виділяється місце для змінних, що використовуються в програмі. Поділ сегментів на сегменти даних, стеку та додаткових даних зумовлена тим, що змінні, які тут визначені мають різні властивості. Змінні в програмі можна розбити на 2 великі групи:

1. Змінні, які явна використовуються в програмі виходячи з її логіки. Такі змінні як правило мають імена і використовуються вони для збереження даних, які мають наперед визначений зміст. Наприклад, змінна що використовується для збереження коду натиснутої клавіші.
2. Часто в програмі необхідно мати місце для збереження тимчасових даних.

### Сегмент даних.

Використовується для збереження змінних, що визначає програміст. Як правило це робиться при допомозі псевдооператорів виділення пам’яті. Для визначення адреси початку сегменту використовується регістр DS. Для визначення другої компоненти адреси (відповідного зміщення всередині сегменту) використовується кілька способів адресації. Зауважимо, що способів визначення адреси змінної набагато більше ніж способів визначення адреси команди. В мові асемблер є засоби для явного розділ. збереження змінних. Ці змінні можуть мати або не мати початкових значень, а також можуть бути змінними константами. Розділяти ці типи змінних не обов’язково. Усіх їх можна зберігати в сегменті даних.

### Ініціалізовані змінні.

Змінні, початкові значення яких відоме як правило записуються в сегменті, що починається директивою **data**. До цих змінних відносяться також ті, початкові значення яких відоме і в процесі роботи вони не повиннVпмінятись.

### Неініціалізовані змінні.

Неініціалізовані змінні записуються в сегмент який починається директивою **data**.

Сегмент стеку використовується для збереження тимчасових даних для яких недоцільно виділяти змінні. Адреса початку цього сегменту розміщується в регістрі SS, а зміщення відносно початку в регістрі SP. Для визначення сегменту стеку використовується директива **stack**. Цей сегмент відрізняється від сегменту даних і коду. В сегменті даних і коду можна явно адресувати б-я комірку пам’яті. Тут значення регістру при виконанні команд. В кожний момент часу пара цих регістрів SS і SP вказує на так звану вершину стеку. Сюди можна записати або прочитати значення. Стек організований за правилом: перший записаний, останній прочитаний. Тому у програмі команди запису і зчитування зі стеку потрібно використовувати попарно. Стек переважно використовується для тимчасового зберігання адреси повернення з програми або перервань. Ще одне ми.За вання стеку – при його допомозі можна передавати аргументи підпрограмам і організовувати локальні змінні.

## Лекція за 1.10.

## Моделі пам’яті

1. **Tiny (файл \*.com).** При цьому регістри CS, DS, SS містять однакові значення. Це найбільш компактна модель пам’яті. Розмір пам’яті не може перевищувати 63Кб. Адресація здійснюється за допомогою зміщення і міток. Оскільки програма на асемблері не є великими, то це не є значним обмеженням. Ця модель широко використовується, особливо в резидентних програмах.
2. **Small**. Сегмент коду відділений від сегменту стеку і сегменту даних. Сегмент даних і стеку об’єднані в єдине. Тому CS і DS мають однакові значення. Це найбільш поширена модель при розробці програм на асемблері. Всі переходи і виклики підпрограм здійснюються за допомогою короткої адресації.
3. **Compact**. Використовується 1 сегмент коду і можливе ми.За вання кількох сегментів даних. тому виклики підпрограм і переходи в програмі є короткими, а доступи до даних – далекими. Сегмент даних і стек об’єднані в одну групу.
4. **Medium.** В значній мірі протилежна до попередніх. Є кілька сегментів коду і один сегмент даних. Тому доступ до підпрограм і переходи в програмі є далекими (потрібна адреса сегменту та зміщення). А доступ до даних здійснюється за допомогою короткої адресації (вказується тільки зміщення).
5. **Large.** Це найбільш загальний спосіб організації пом’яті. Тут можуть використовуватись декілька сегментів коду і даних. Доступ до змінних, переходи на мітки здійснюються за допомогою адреси сегментів та зміщень.
6. **Huge, Flat.** Це досить поширені моделі. Така ж організація пам’яті, як і в **large**. Вона використовується для поєднання асемблера змовами високого рівня. В них можна задати таку модель пам’яті, з’являється можливість роботи з даними що займають більше 64 Кб. В програмі на асемблері таку модель використовувати недоцільно. Модель **Flat** відповідає моделі **Small**, різниця в тому, що використовуються 32 розрядні регістри. Таку модель можна організувати використовуючи процесор вище **і80386.** Переважно її використовують для роботи в захащеному режимі. В цьому випадку доступ до всієї області пам’яті здійснюються за допомогою одного сегменту.

При програмуванні на асемблері краще використовувати перші 2 моделі пам’яті. Можна використовувати інші моделі, але в цих випадках програма ускладнюється, адже доводиться слідкувати за розміщенням кожного сегменту. Для визначення моделі пам’яті використовується директива **model**, що має такий формат запису:

Model [ < модифікатор >] < тип >.

В цій директиві може бути вказано ще ряд параметрів. Єдиним необхідним параметром є параметр <тип>. Тип можна додатково модифікувати. Для цього необхідно вказати один із 4 модифікаторів:а**nearstack; farstack; use16; use32.** При першому стек і дані розташинані в межах одного сегменту; **farstack** вказує, що сегмент даних буде розташинаний за межами стеку. **Nearstack** використовується по замовчуванню у всіх стандартних моделях пам’яті. Він дозволяє організувати інтерфейс з мовами високого рівня. Тут глобальні дані розміщені в сегменті даних, а локальні – в сегменті стеку. Два останні модифікатори використовуються тільки для МП 80386 і старше, вони вказують які сегменти використовуються в програмі, 16-розрядні (64 Кб сегмент) і 32-розрядні (4Гб сегмент). Доцільно використовувати ці модифікатори при розробці програм при використанні 32-розрядних компіляторів.

## Псевдооператори (директиви).

Псевдооператори керують роботою асемблера, а не МП. Вони дозволяють визначати сегменти і процедури (надавати імена програмам і елементам даних, резервувати області пом’яті та виконувати багато інших задач). Псевдооператори мають таку загальну форму запиту:а

[<ідентифікатор>] псевдооператор [<операнд>][; коментар].

Обов’язковим є тільки поле псевдооператори. Для одних псевдооператорів ідентифікатор обов’язковий, для інших заборонений, для третіх необов’язковий, це ж відноситься і до операнди, коментарі завжди необов’язкові. Псевдооператор може починатися в будь-якому місці рядка, він повинен бути віддаленим від попереднього поля хоча б одним пропуском. Ми розглянемо найбільш розповсюджені псевдооператори, їх можна поділити на дві групи: псевдооператори даних і псевдооператори керування лістингом. Псевдооператори даних можна розділити на 5 груп:а

1. Псевдооператори визначення ідентифікаторів. Вони дозволяють виразу присвоїти символічне ім’я, тобто ідентифікатор. В якості виразу може бути константа, адреса, інше символічне ім’я. Після такої директиви цей ідентифікатор можна використовувати в будь-якому місці де необхідно вказати вираз. Два оператори:а**EQU** і **=**, ці псевдооператори подібні, але між ними є істотна різниця:а

Позначені знаком “=” ідентифікатори можна перевизначати, а визначені псевдооператором **EQU** неможна. Псевдооператор **EQU** можна використовувати як з числовими такі з текстовими виразами, а псевдооператор “=” тільки з числовими. Приклади:

**К1аEQU 1024 –** присвоєння імені константі; **T\_1аEQU DS:[BP][SI].** Комбінації адрес **DS:[BP][SI]** присвоюється символічне ім’я, всюди в програмі замість комбінації цих адрес можна використовувати **T\_1**. **В\_Т EQU B\_T1**, визначення синоніму, тобто ім’я **В\_Т** буде мати те ж значення як і зміна **В\_Т1**. **REG1аEQU CX** – присвоєння імені регістру. При використанні псевдооператора “=” в якості виразу можуть виступати математичні перетворення. В директиві **EQU** також можуть бути прості математичні перетворення, вони будуть виконані асемблером під час трансляції. Наприклад: **CON1аEQU CON2\*2 CONS1=24 CONS1=29 CONS1=CONS1+1.**

1. Псевдооператори визначення даних. В асемблері можна використовувати комірки пам’яті для збереження поіменованих даних, тобто змінних. Найбільш використовуються такі псевдооператори для резервування пам’яті:1) DB – Define Byte; 2) DW – Define Word; 3) DD – Define Double Word; 4) DQ; 5) DF; 6) DT. В основному вони розрізняються по об’єму пам’яті, який резервується. Псевдооператори визначення даних мають такий формат запису: **ім’я псевдооператор вираз [,……].** В якості виразу найчастіше може бути константа. Псевдооператор DB резервує пам’ять об’ємом 1 байт. DW – об’ємом одне слово. DD – подвійне слово. DQ – 2пподвійні слова. DF – 6 байт. DT – 10 байт. Наприклад змінна xyz DW 2S – резервує для змінної ( ). Треба пам’ятати про макс. Vпмін. значення даних, які можуть бути розміщені у відведеній пам’яті в кожному з цих випадків, так наприклад: макс. значення байта без знака не може перевищувати 255, макс. значення байта зі знаком 127, а мін. – 128. Псевдооператори можна використовувати для створення в пам’яті таблиць, наприклад: **А\_ТАВ DB 3, 2, 4, 6, 8, 0, 0, 0**

**DB 5, 4, 2, 1**

Такий псевдооператор **DB** резервує в пам’яті місце для 12 значень елементів таблиці **А\_ТАВ** кожний розміром в байт. Можна вказувати довільну кількість елементів таблиці рядку, важливо, щоб вони помістились в 132 позиції рядка. В асемблері є операція **DUP**, яка дозволяє скорочувати задання таблиць у випадку однакових елементів. Даний псевдооператор можна записати так:

**А\_ТАВ DB 3, 2, 4, 6, 8, 3 DUP (0),**

**DB 5, 4, 2, 1**

В пам’яті можна резервувати пам’ять для змінної, але конкретного значення їй не надавати. Для цього в полі виразу треба вказати знак питання. Приклад: **X\_1аDB ? X\_2 DW ?**. Такі псевдооператори резервують в пам’яті місце байту і слова, але початкового значення їм не надають. **X\_1аi X\_2** при цьому не будуть містити ніяких специфічних значень, навіть **(0)**. Можна зарезервувати в пам’яті місце для таблиці. Наприклад: **А\_ТАВ DB 12 DUP (3)** резервує місце для таблиці з 12 елементів. Можна в якості значення змінної задавати рядок символів, для цього текст заключається в апострофи.

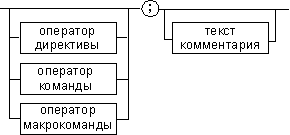
1. Псевдооператори визначення сегменту і процедури. Для визначення сегменту використовуються псевдооператори: **SEGMENTаi ENDS**. Це псевдооператори ділять вихідну програму на сегменти. Вони відповідно відмічають початок і кінець сегменту, але вини не повідомляють асемблер якого роду сегмент повинен бути визначений. Для цього використовується псевдооператор **ASSUME,** він має таку форму запису: **ASSUME** регістр сегмента: ім’я сегмента [,…….]. це регістр сегмента – це ім’я одного із сегментних регістрів **CS, DS, SS, ES**. Якщо П 80386 і вище то там ще буде **FS, GS**. Ім’я сегмента це ім’я яке вказується в псевдооператорі **SEGMENT**. Цей псевдооператор допомагає асемблеру перетворювати мітки в адреси, він повідомляє, яким регістром хочете скористяватися при адресації міток. Наприклад: **ASSUME DS: DSEG –** означає, якщо при трансляції програми буде згадана мітка із сегмента **DSEG**, тобто регістр **DS** буде завжди вказувати на початок сегмента **DSEG**. Майже завжди його поміщають після псевдооператора **SEGMENT.** Зауважимо, що адресу початку сегменту потрібно явно завантажити в **DS**. Псевдооператори **PROG** і **ENDP** визначають початок і кінець процедури. Кожна процедура повинн0ппочинатись

## Структура программы на ассемблере Синтаксис ассемблера Стандартные директивы сегментации Упрощенные директивы сегментации

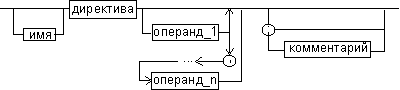
Программа на ассемблере представляет собой совокупность блоков памяти, называемых *сегментами памяти*. Программа может состоять из одного или нескольких таких блоков-сегментов. Каждый сегмент содержит совокупность предложений языка, каждое из которых занимает отдельную строку кода программы. Предложения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

*команды или инструкции* представляющие собой символические аналоги машинных команд.   
В процессе трансляции инструкции ассемблера преобразуются в соответствующие команды системы команд микропроцессора; **макрокоманды** - оформляемые определенным образом предложения текста программы, замещаемые во время трансляции другими предложениями; **директивы,** являющиеся указанием транслятору ассемблера на выполнение некоторых действий. У директив нет аналогов в машинном представлении; **строки комментариев**, содержащие любые символы, в том числе и буквы русского алфавита. Комментарии игнорируются транслятором.

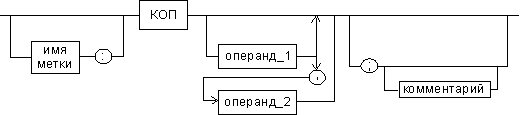
### Синтаксис ассемблера Предложения, составляющие программу, могут представлять собой синтаксическую конструкцию, соответствующую команде, макрокоманде, директиве или комментарию. Для того чтобы транслятор ассемблера мог распознать их, они должны формироваться по определенным синтаксическим правилам. 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 диаграммы*. К примеру, синтаксис предложений ассемблера можно описать с помощью синтаксических диаграмм, показанных на следующих рисунках.



***Рис. 1.*** *Формат предложения ассемблера*



***Рис. 2.*** *Формат директив*



***Рис. 3.*** *Формат команд и макрокоманд*

На этих рисунках:

**имя метки** - идентификатор, значением которого является адрес первого байта того предложения исходного текста программы, которое он обозначает; им*я* - идентификатор, отличающий данную директиву от других одноименных директив. В результате обработки ассемблером определенной директивы этому имени могут быть присвоены определенные характеристики; **код операции** *(КОП) и директива* - это мнемонические обозначения соответствующей машинной команды, макрокоманды или директивы транслятора; **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. **Лексемам*и* являются**: **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 может 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**;**цепочки символов** - последовательности символов, заключенные в одинарные или двойные кавычки; **целые числа** в одной из следующих систем счисления: *двоичной, десятичной, шестнадцатеричной*. Отождествление чисел при записи их в программах на ассемблере производится по определенным правилам: **Десятичные числа** не требуют для своего отождествления указания каких-либо дополнительных символов, например 25 или 139. Для отождествления в исходном тексте программы **двоичных чисел** необходимо после записи нулей и единиц, входящих в их состав, поставить латинское "**b**", например 10010101**b**. **Шестнадцатеричные числа** имеют больше условностей при своей записи: *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**. *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 ноль: **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. Возм**ожно провести следующую классификацию операндов:**

постоянные, или непосредственные, операнды ,адресные операнды .перемещаемые операнды ,счетчик адреса ,регистровый операнд .базовый и индексный операнды .структурные операнды ,Записи ,Рассмотрим подробнее характеристику операндов из приведенной классификации: **Постоянные или непосредственные операнды** - число, строка, имя или выражение, имеющие некоторое фиксированное значение. Имя не должно быть перемещаемым, то есть зависеть от адреса загрузки программы в память. К примеру, оно может быть определено операторами **equ** или **=**.

num     equ     5

imd = num-2

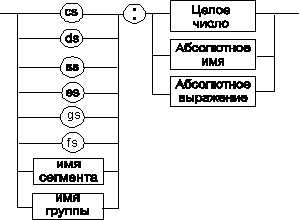
        mov     al,num  ;эквивалентно mov al,5

;5 здесь непосредственный операнд

        add     [si],imd        ; imd=3 - непосредственный операнд

        mov     al,5            ;5 - непосредственный операнд

В данном фрагменте определяются две константы, которые затем используются в качестве непосредственных операндов в командах пересылки mov и сложения add.  *Адресные операнды* - задают физическое расположение операнда в памяти с помощью указания двух составляющих адреса: *сегмента* и *смещения* (рис. 4).



* ***Рис. 4.*** *Синтаксис описания адресных операндов*

К примеру:

mov     ax,0000h

        mov     ds,ax

        mov     ax,ds:0000h     ;записать слово в ax из области памяти по

                                        ;физическому адресу

0000:0000

* Здесь третья команда mov имеет адресный операнд.
* *Перемещаемые операнды* - любые символьные имена, представляющие некоторые адреса памяти. Эти адреса могут обозначать местоположение в памяти некоторых инструкции (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.

К примеру:

data    segment

mas\_w   dw      25 dup (0)

:

code    segment

:

        lea     si,mas\_w        ;mas\_w - перемещаемый операнд

* В этом фрагменте *mas\_w* - символьное имя, значением которого является начальный адрес области памяти размером 25 слов. Полный физический адрес этой области памяти будет известен только 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 листинге колонка с уровнем вложенности). Если взять в качестве пример 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. В качестве примера использования в команде значения счетчика адреса можно привести следующий:

jmp $+3 ;безусловный переход на команду mov

 cld ;длина команды cld составляет 1 байт

 mov al,1

  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 памяти. В большинстве случаев эту информацию можно получить, зная формат машинной команды и анализируя колонку листинга с объектным кодом команды.  ***Регистровый операнд*** - это просто имя регистра. В программе на ассемблере можно использовать имена всех регистров общего назначения и большинства системных регистров.

mov al,4 ;константу 4 заносим в регистр al

 mov dl,pass+4 ;байт по адресу pass+4 в регистр

dl

 add al,dl ;команда с регистровыми

операндами

* *Базовый и индексный операнды*. Этот тип операндов используется для реализации [косвенной базовой](http://swizard.chat.ru/asmtutorial/Les_06.htm#КосвеннаяБазовая), [косвенной индексной адресации](http://swizard.chat.ru/asmtutorial/Les_06.htm#КосвеннаяБазовая) или их [комбинаций](http://swizard.chat.ru/asmtutorial/Les_06.htm#КосвеннаяБазовая) и [расширений](http://swizard.chat.ru/asmtutorial/Les_06.htm#КосвеннаяБазовая).
* *Структурные операнды* используются для доступа к конкретному элементу сложного типа данных, называемого [*структурой*](http://swizard.chat.ru/asmtutorial/Data.htm#ОпределениеСтруктуры).
* *Записи* (аналогично структурному типу) используются для доступа к битовому полю некоторой [записи](http://swizard.chat.ru/asmtutorial/Data.htm#ОпределениеЗаписи).

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.

[Арифметические операторы](http://swizard.chat.ru/asmtutorial/#АрифметическиеOператоры) [Операторы сдвига](http://swizard.chat.ru/asmtutorial/#ОператорыСдвига) [Операторы сравнения](http://swizard.chat.ru/asmtutorial/#ОператорыСравнения) [Логические операторы](http://swizard.chat.ru/asmtutorial/#ЛогическиеОператоры) [Индексный оператор](http://swizard.chat.ru/asmtutorial/#ИндексныйОператор) [Оператор переопределения типа](http://swizard.chat.ru/asmtutorial/#ОператорПереопределенияТипа) [Оператор переопределения сегмента](http://swizard.chat.ru/asmtutorial/#ОператорПереопределенияСегмента) [Оператор именования типа структуры](http://swizard.chat.ru/asmtutorial/#ОператорИменованияТипа) [Оператор получения сегментной составляющей адреса выражения](http://swizard.chat.ru/asmtutorial/#SEG) [Оператор получения смещения выражения](http://swizard.chat.ru/asmtutorial/#Offset)

В [*табл. 2*](http://swizard.chat.ru/asmtutorial/#Табл_2) приведены поддерживаемые языком ассемблера операторы и перечислены их приоритеты. Дадим краткую характеристику операторов:

* *Арифметические операторы*. К ним относятся:
  + унарные "**+**" и "**-**";
  + бинарные "**+**" и "**-**";
  + умножения "**\***";
  + целочисленного деления "**/**";
  + получения остатка от деления "**mod**".

Эти операторы расположены на уровнях приоритета 6, 7, 8 в [*табл. 2*](http://swizard.chat.ru/asmtutorial/#Табл_2). Например,

tab\_size equ 50 ;размер массива в байтах

size\_el equ 2 ;размер элементов

:

;вычисляется число элементов массива и заносится в

регистр cx

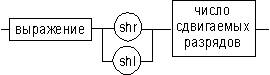
 mov cx,tab\_size / size\_el ;оператор "/"

* ***Рис. 5.*** *Синтаксис арифметических операций*
* *Операторы сдвига* выполняют сдвиг выражения на указанное количество разрядов (рис. 6). Например,

mask\_b equ 10111011

:

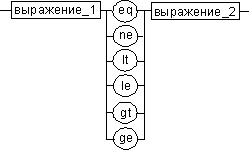
mov al,mask\_b shr 3 ;al=00010111



* ***Рис. 6.*** *Синтаксис операторов сдвига*
* *Операторы сравнения* (возвращают значение "истина" или "ложь") предназначены для формирования логических выражений (см. рис. 7 и [*табл. 1*](http://swizard.chat.ru/asmtutorial/#Табл_1)). Логическое значение "истина" соответствует цифровой единице, а "ложь" - нулю. Например,

|  |
| --- |
| tab\_size equ 30 ;размер таблицы  :   mov al,tab\_size ge 50 ;загрузка размера  таблицы в al   cmp al,0 ;если tab\_size < 50, то   je m1 ;переход на m1  :  m1: : |

* В этом примере если значение tab\_size больше или равно 50, то результат в al равен 0ffh, а если tab\_size меньше 50, то al равно 00h. Команда cmp сравнивает значение al с нулем и устанавливает соответствующие флаги в flags/eflags. Команда je на основе анализа этих флагов передает или не передает управление на метку m1.



* ***Рис. 7.*** *Синтаксис операторов сравнения*

##### Таблица 1. Операторы сравнения

|  |  |
| --- | --- |
| Оператор | Значение |
| eq | ИСТИНА, если выражение\_1 равно выражение\_2 |
| ne | ИСТИНА, если выражение\_1 не равно выражение\_2 |
| lt | ИСТИНА, если выражение\_1 меньше выражение\_2>ИСТИНА, если выражение\_1 не равно выражение\_2 |
| le | ИСТИНА, если выражение\_1 меньше или равно выражение\_2 |
| gt | ИСТИНА, если выражение\_1 больше выражение\_2 |
| ge | ИСТИНА, если выражение\_1 больше или равно выражение\_2 |

* *Логические операторы* выполняют над выражениями побитовые операции (рис. 8). Выражения должны быть абсолютными, то есть такими, численное значение которых может быть вычислено транслятором. Например:

|  |
| --- |
| flags   equ     10010011           mov    al,flags xor 01h        ;al=10010010;пересылка в al поля flags с                                          ;инвертированным пр  авым битом |



* ***Рис. 8.*** *Синтаксис логических операторов*
* *Индексный оператор* **[ ]**. Не удивляйтесь, но 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. 9). Например,

|  |
| --- |
| mov     ax,mas[si]      ;пересылка слова по адресу mas+(si) в регистр ax |



* ***Рис. 9.*** *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.   
  К примеру, в нашем случае запись в комментариях последнего фрагмента программы mas + (si) означает вычисление следующего выражения: значение смещения символического имени mas плюс содержимое регистра si.
* *Оператор переопределения типа* **ptr** применяется для переопределения или уточнения типа метки или переменной, определяемых выражением (рис. 10).   
  Тип может принимать одно из следующих значений: **byte, word, dword, qword, tbyte, near, far**. Например,

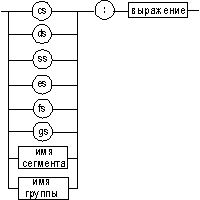
|  |
| --- |
| d\_wrd   dd      0  ...          mov     al,byte ptr d\_wrd+1 ;пересылка второго байта из двойного слова |

* Поясним этот фрагмент программы. Переменная *d\_wrd* имеет тип двойного слова. Что делать, если возникнет необходимость обращения не ко всей переменной, а только к одному из входящих в нее байтов (например, ко второму)? Если попытаться сделать это командой   
  mov al,d\_wrd+1, то транслятор выдаст сообщение о несовпадении типов операндов. Оператор **ptr** позволяет непосредственно в команде переопределить тип и выполнить команду.



* ***Рис. 10.*** *Синтаксис оператора переопределения типа*
* *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. 11). Этот момент очень важен, поэтому поясню его подробнее. 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? Посмотрите раздел [*"Программная модель микропроцессора"*](http://swizard.chat.ru/asmtutorial/CPUmodel.htm), там, где мы описывали назначение регистров общего назначения.   
  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*. Что подразумевает термин *"по умолчанию"*? Вспомните *"рефлексы"*, о которых мы говорили на уроке 1. 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 же, нельзя, но можно чуть-чуть подкорректировать. Делается это с помощью необязательного поля *префикса машинной команды* (см. [формат машинной команды](http://swizard.chat.ru/asmtutorial/Les_06.htm#ФорматМашиннойКоманды)). 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 (переопределения) 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. Например,

|  |
| --- |
| .code  ...          jmp     met1    ;обход обязателен, иначе поле ind  будет трактоваться ;как очередная команда  ind     db      5       ;описание поля данных в сегменте команд  met1:  ...  mov al,cs:ind ;переопределение сегмента позволяет работать с                          ;данными, определенными внутри сегмента кода |



* ***Рис. 11.*** *Синтаксис оператора переопределения сегмента*
* *Оператор именования типа* [структуры](http://swizard.chat.ru/asmtutorial/Data.htm#ОпределениеСтруктуры) **.** (точка) также заставляет транслятор производить определенные вычисления, если он встречается в выражении.
* *Оператор получения сегментной составляющей адреса выражения* **seg** возвращает физический адрес сегмента для выражения (рис. 12), в качестве которого могут выступать метка, переменная, имя сегмента, имя группы или некоторое символическое имя.



***Рис. 12.*** *Синтаксис оператора получения сегментной составляющей*

* *Оператор получения смещения выражения* **offset** позволяет получить значение смещения выражения (рис. 13) в байтах относительно начала того сегмента, в котором выражение определено.



***Рис. 13.*** *Синтаксис оператора получения смещения*

Например,

|  |
| --- |
| .data  pole    dw      5  ...  .code  ...          mov     ax,seg  pole          mov     es,ax          mov     dx,offset       pole    ;теперь в паре es:dx полный адрес pole |

Как и в языках высокого уровня, выполнение операторов ассемблера при вычислении выражений осуществляется в соответствии с их приоритетами (см. табл. 2). Операции с одинаковыми приоритетами выполняются последовательно слева направо. Изменение порядка выполнения возможно путем расстановки круглых скобок, которые имеют наивысший приоритет.

##### Таблица 2. Операторы и их приоритет

|  |  |
| --- | --- |
| Оператор | Приоритет |
| length, size, width, mask, (, ), [, ], <, > | 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 |

### Директивы сегментации

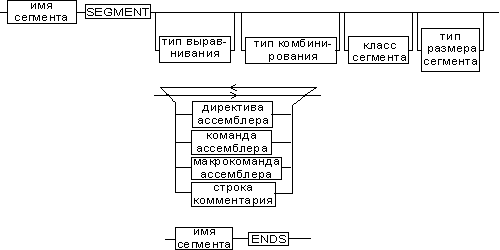
В ходе предыдущего обсуждения мы выяснили все основные правила записи команд и операндов в программе на ассемблере. Открытым остался вопрос о том, как правильно оформить последовательность команд, чтобы транслятор мог их обработать, а микропроцессор - выполнить.

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

* с одним сегментом кода;
* с одним сегментом стека;
* с одним сегментом данных;
* 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 рис. 14:



***Рис. 14.*** *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.

* ***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 i80х86 выполняется быстрее. Допустимые значения этого атрибута следующие:
  + **BYTE** - выравнивание не выполняется. Сегмент может начинаться с любого адреса памяти;
  + **WORD** - сегмент начинается по адресу, кратному двум, то есть последний (младший) значащий бит физического адреса равен 0 (выравнивание на границу слова);
  + **DWORD** - сегмент начинается по адресу, кратному четырем, то есть два последних (младших) значащих бита равны 0 (выравнивание на границу двойного слова);
  + **PARA** - сегмент начинается по адресу, кратному 16, то есть последняя шестнадцатеричная цифра адреса должна быть 0h (выравнивание на границу параграфа);
  + **PAGE** - сегмент начинается по адресу, кратному 256, то есть две последние шестнадцатеричные цифры должны быть 00h (выравнивание на границу 256-байтной страницы);
  + **MEMPAGE** - сегмент начинается по адресу, кратному 4 Кбайт, то есть три последние шестнадцатеричные цифры должны быть 000h (адрес следующей 4-Кбайтной страницы памяти).

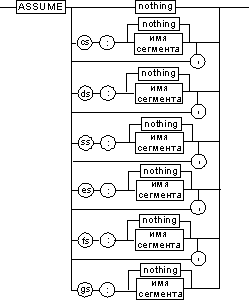
По умолчанию тип выравнивания имеет значение **PARA**.

* ***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:
  + **PRIVATE** - сегмент не будет объединяться с другими сегментами с тем же именем вне данного модуля;
  + **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;
  + **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;
  + **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;
  + **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**.

* ***Атрибут класса сегмента*** (тип класса) - это заключенная в кавычки строка, помогающая компоновщику определить соответствующий 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;
* ***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:
  + **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;
  + **USE32** - сегмент будет 32-разрядным. При формирования 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. 15. 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 аннулируются, 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. 15).



***Рис. 15.*** *Директива ASSUME*

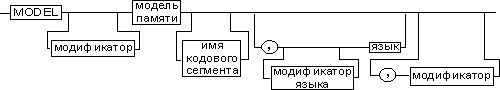
На уроке 3 мы рассматривали пример программы с директивами сегментации. Эти директивы изначально использовались для оформления программы в трансляторах MASM и TASM. Поэтому их называют *стандартными директивами сегментации*.

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 комбинированием 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).

В листинге 1 приведен пример программы с использованием упрощенных директив сегментации:

|  |
| --- |
| Листинг 1. Использование упрощенных директив сегментации  ;---------Prg\_3\_1.asm-------------------------------  masm                    ;режим работы TASM: ideal или masm  model   small           ;модель памяти  .data                   ;сегмент данных  message db      'Введите две шестнадцатеричные цифры,$'  .stack                  ;сегмент стека          db      256     dup ('?')       ;сегмент стека  .code                   ;сегмент кода  main    proc            ;начало процедуры main          mov     ax,@data        ;заносим адрес сегмента данных в регистр ax          mov     ds,ax   ;ax в ds  ;далее текст программы (см. сегмента кода в листинге 3.1 книги)          mov     ax,4c00h        ;пересылка 4c00h в регистр ax          int     21h             ;вызов прерывания с номером 21h  main    endp            ;конец процедуры main  end     main            ;конец программы с точкой входа main |

Синтаксис директивы **MODEL** показан на рис. 16.



***Рис. 16.*** *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 табл. 3.

##### Таблица 3. Упрощенные директивы определения сегмента

|  |  |  |
| --- | --- | --- |
| Формат директивы   (режим MASM) | Формат директивы   (режим IDEAL) | Назначение |
| .CODE [имя] | CODESEG[имя] | Начало или продолжение сегмента кода |
| .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 (см. [*табл. 5*](http://swizard.chat.ru/asmtutorial/#Табл_5)). Перечислим эти идентификаторы и их значения (табл. 4). Таблица 4. Идентификаторы, создаваемые директивой MODEL

|  |  |
| --- | --- |
| 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 |

Если вы посмотрите на текст [*листинга 1*](http://swizard.chat.ru/asmtutorial/#Листинг_1), то увидите пример использования одного из этих идентификаторов. Это **@data**; с его помощью мы получили значение физического адреса сегмента данных нашей программы.

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 табл. 5 приведены некоторые значения параметра *модель памяти* директивы **MODEL**.  Таблица 5. Модели памяти

|  |  |  |  |
| --- | --- | --- | --- |
| 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 |
| 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. 6). Таблица 6. Модификаторы модели памяти

|  |  |
| --- | --- |
| 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. Необходимость 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.