МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«*Київський політехнічний інститут*»

СИСТЕМНЕ ПРОГРАМУВАННЯ

Методичні вказівки до виконання курсової роботи

для студентів напряму підготовки 6.050102 «Комп'ютерна інженерія»

Ухвалено
Вченою радою ФПМ
НТУУ «КПІ»
Протокол № _ від __.__.2014 р.

Київ

НТУУ «КПІ»

2014

Системне програмування: методичні вказівки до виконання курсової роботи з дисципліни «Системне програмування» для студентів напряму підготовки «Комп'ютерна інженерія» [Електронне видання] / О.К.Тесленко, І.П.Дробязко. – К. : НТУУ «КПІ», 2014. – 66 с.

Навчально-методичне видання

СИСТЕМНЕ ПРОГРАМУВАННЯ

Методичні вказівки до виконання курсової роботи
для студентів напряму підготовки 6.050102 «Комп'ютерна інженерія»

Методичні вказівки розроблено для виконання курсової роботи з дисципліни «Системне програмування», а саме створення компілятора для програм мовою Асемблера. Методичні вказівки містять вимоги до роботи та оформлення її результатів, опис методики виконання завдання з прикладами та додаткові інформаційні матеріали. Навчальне видання призначене для студентів, які навчаються за напрямами 6.050102 «Комп'ютерна інженерія» факультету прикладної математики НТУУ «КПІ»

Укладачі *Тесленко Олександр Кирилович,* канд. техн. наук, доц.
Дробязко Ірина Павлівна, ст. викл.

Відповідальний
за випуск *Тарасенко Володимир Петрович*, д-р техн. наук, проф.

Рецензент *Темнікова Олена Леонідівна*, ст. викл.

ЗМІСТ

ВСТУП

Дисципліна «Системне програмування» є спеціальною дисципліною циклу професійної та практичної підготовки бакалаврів за напрямом 6.050102 «Комп'ютерна інженерія», яка ознайомлює студентів з проблемами, які стоять перед системним програмуванням, і основними напрямками їх вирішення. Одним з важливих видів індивідуальних завдань, що передбачені навчальною програмою дисципліни «Системне програмування» та виконуються студентами при її вивченні, є курсова робота (КР).

КР є творчим рішенням конкретної задачі, а саме розробки компілятора програм мовою Асемблера, виконуваним студентом самостійно під керівництвом викладача згідно із завданням, на основі набутих знань та умінь з даної дисципліни та суміжних дисциплін: «Основи програмування», «Структури даних та алгоритми», «Комп'ютерна логіка. Комп'ютерна арифметика», «Об'єктно-орієнтоване програмування». Виконання курсової роботи сприяє розширенню та поглибленню отриманих теоретичних знань щодо архітектури процесорів, машинної мови та машинно-орієнтованої мови програмування Асемблера та ін., формуванню вмінь їх практичного використання, самостійного вирішення відповідних інженерних завдань програмним шляхом, розвитку навичок тестування та документування розробленого програмного продукту.

Методичні вказівки містять загальну постановку завдання та зміст індивідуального завдання, вказівки щодо методики реалізації завдання з прикладами та додатковими інформаційними матеріалами, вимоги до виконання завдання, оформлення звіту та захисту курсової роботи тощо.

1. МЕТА ТА ЗАВДАННЯ КУРСОВОЇ РОБОТИ

Метою курсової роботи з дисципліни «Системне програмування» є закріплення та поглиблення знань щодо мови Асемблера та її взаємозв'язку з архітектурою процесора, набуття практичних навичок у реалізації компіляторів програм мовою Асемблера.

Завданням даної курсової роботи є розробка компілятора програм, написаних мовою Асемблера. В результаті виконання **курсової роботи** студент має:

ЗНАТИ:

- особливості архітектури процесорів фірми Intel, внутрішнє представлення та формати інструкцій і даних; відповідність машинно-орієнтованої мови програмування Асемблера машинній мові та ін.;
- загальну методику побудови компіляторів (лексичний аналіз, синтаксичний аналіз, обробка лексем, формування коду, виявлення помилок тощо);

УМІТИ:

- самостійно працювати з навчально-методичною, технічною, періодичною літературою й іншими інформаційними джерелами за тематикою розробки;
- практично використовувати отримані знання для самостійного вирішення завдання програмним шляхом і побудови компілятора;
- аналізувати та алгоритмізувати поставлену задачу, обґрунтовувати структурну організацію проекту, оцінювати

доцільність застосування обраних програмних засобів для створення програмного продукту;

- документувати новостворений програмний код, описувати структуру програми та схему взаємодії її складових;
- розробляти тестові модулі та використовувати їх для тестування розробленого програмного продукту;

НАПРАЦЮВАТИ ДОСВІД:

- роботи в різних середовищах розробки та відлагодження програм;
- створення завершених програмних продуктів.

2. ЗАВДАННЯ НА КУРСОВУ РОБОТУ

Тема курсової роботи – Розробка компілятора програм мовою Асемблера.

Основним завданням і результатом роботи програми компілятора повинно бути створення для програм мовою Асемблера текстового файла лістинга (розширення .lst), подібного до файла лістинга компілятора MASM або TASM.

Враховуючи, що створення компілятора є трудомістким процесом, який потребує значних витрат часу, у варіантах завдань на курсову роботу використовуються суттєві обмеження на перелік допустимих машинних інструкцій, режимів адресації даних і команд та допустимих директив. Будь який із запропонованих студентам варіантів індивідуальних завдань фактично вказує на підмножину стандартної мови Асемблера процесорів фірми Intel. Тим не менш, цієї підмножини достатньо для забезпечення мети розробки.

Нижче наведені два приклади індивідуальних завдань на розробку компілятора, що визначають вищезазначені обмеження вимог до компілятора.

Приклад індивідуального завдання 1

Ідентифікатори

Містять великі та малі букви латинського алфавіту і цифри. Починаються з букви. Великі та малі букви не відрізняються.

Довжина ідентифікаторів – не більше 8 символів.

Константи

Шістнадцяткові, десяткові, двійкові та текстові константи.

Директиви

END, SEGMENT без операндів, ENDS, ASSUME.

DB, DW, DD з одним операндом - константою (рядкові константи тільки для DB).

Розрядність даних та адрес

16-розрядні дані та зміщення у сегменті, у випадку 32-розрядних даних та 32-розрядних зміщень генеруються відповідні префікси зміни розрядності.

Адресація операндів пам'яті

Індексна адресація (Val1[si],Val1[bx],Val1[eax],Val1[edi] і т.п.)

Заміна сегментів

Префікси заміни сегментів можуть задаватись явно, а при необхідності автоматично генеруються компілятором

Машинні команди

Cli

Inc **reg**

Dec **mem**

Add **reg, reg**

Cmp **reg, mem**

Xor **mem, reg**

Mov **reg, imm**

Or **mem, imm**

Jb

Jmp (внутрішньо сегментна відносна адресація)

де **reg –** 8, 16 або 32-розрядні регістри загального призначення (РЗП);

mem – адреса операнда в пам’яті;

imm - 8, 16 або 32-розрядні безпосередні дані (константи).

Приклад індивідуального завдання 2

Ідентифікатори

Містять великі та малі букви латинського алфавіту і цифри. Починаються з букви. Великі та малі букви не відрізняються.

Довжина ідентифікаторів – не більше 6 символів.

Константи

Десяткові цілі та десяткові дробові (наприклад 3.14 і т.п.) константи.

Директиви

END, SEGMENT без операндів, ENDS,

програма може мати тільки один сегмент кодів і тільки один сегмент даних.

DD, DQ з одним операндом - константою. Для директив DD та DQ можуть задаватись десяткові дробові константи.

Десяткові дробові константи в директиві DD транслюються у формат короткого дійсного числа, а в директиві DQ – у формат довгого дійсного.

Розрядність даних та адрес

32-розрядні дані та зміщення в сегменті, 16-розрядні дані та зміщення не використовуються.

Адресація операндів пам'яті

Базова індексна адресація ([edx+esi],[ebx+ecx] і т.п.) з оператором визначення типу ptr при необхідності

Заміна сегментів

Префікси заміни сегментів можуть задаватись тільки явно.

Машинні команди

Fxtract

Fsubr **mem**

Fsubrp **st(j),st(0)**

Fucomp **st(j)**

де **mem –** адреса операнда в пам’яті;

st(j) - регістр співпроцесора (j=0,1,...,7).

Завдання на виконання КР студенти отримують на початку семестра, в якому передбачена курсова робота. В ньому сформульована загальна постановка завдання, індивідуальне завдання, визначені основні вихідні дані та вимоги до роботи і очікувані результати, а також календарний план-графік виконання курсової роботи. В окремих випадках пункти завдання (або завдання в цілому) можуть бути змінені студентом за згодою викладача.

3. СКЛАД, ОБСЯГ І СТРУКТУРА КУРСОВОЇ РОБОТИ

3.1. Вихідні дані та загальні вимоги до розробки

1. *Вихідні дані* компілятора – текстовий файл з довільною програмою мовою Асемблера, яка складена у відповідності з обмеженнями індивідуального варіанта курсової роботи.

Для підготовки тексту програми мовою Асемблера може використовуватися довільний редактор (наприклад, стандартний додаток OS Windows Notepad).

2. *Результатом роботи* компілятора повинно бути створення текстового файла лістинга (розширення .lst).

Формат файла лістинга має співпадати з форматом файла лістинга компілятора MASM або TASM. Діагностичні повідомлення формуються українською мовою. Таблиця символів у файлі лістинга може бути представлена у довільному форматі.

3. *Імена початкового асемблерного файла* для обробки компілятором та створюваного файла лістинга повинні задаватися у командному рядку при запуску програми компілятора.
4. *Усі діагностичні повідомлення*, що міститиме файл лістинга, повинні також виводитися на екран монітора. Крім того, на екран повинна виводитися загальна кількість помилок, виявлених у початковій програмі.
5. *На усі синтаксичні конструкції* (ідентифікатори, константи, директиви, машинні команди, режими адресації і т.д.), які допускаються в компіляторі TASM (MASM), проте виходять за рамки обмежень варіанта курсової роботи, повинно видаватись *діагностичне повідомлення про синтаксичну помилку*.

При здійсненні розробки компілятора повинні бути вирішені наступні задачі:

1. Створення тестових програм мовою Асемблера, що відповідають вимогам індивідуального завдання та дозволяють перевірити коректність роботи компілятора:
   - без помилок і з обмеженнями індивідуального завдання;
   - з помилками, а також з ігноруванням обмежень івіндивідуального завдання.
   - створення аналогічних тестових програм і отримання для них за допомогою компілятора MASM або TASM файлів лістингів

для подальшого їх порівняння з результатами роботи розробленого компілятора.

2. Створення лексичного аналізатора програми мовою Асемблера.
3. Створення програми 1-го перегляду (формування таблиці ідентифікаторів, визначення кількості байтів, які будуть формуватися за кожною інструкцією).
4. Створення програми 2-го перегляду (генерування команд та даних, формування файла лістинга).

3.2. Вимоги до мови програмування

Рекомендується створювати модулі компілятора мовою Паскаль, Delphi або C++ з можливим і обґрунтованим використанням Асемблерних вставок.

3.3. Вимоги до тестових файлів

Тестовий файл є початковою програмою мовою Асемблера зі структурою, яка відповідає вимогам до початкових програм – мати два або більше (у відповідності до конкретного завдання) логічних сегментів та закінчуватись директивою END.

Сукупність рядків в межах логічних сегментів повинна відповідати лише правилам синтаксису відповідних директив чи машинних інструкцій Асемблера згідно із завданням. **Реалізація того чи іншого алгоритму у такій початковій програмі не передбачається і не рекомендується.**

У тестовій початковій програмі необхідно забезпечити перевірку виконання кожного із елементів конкретного завдання, зокрема, кожна із заданих в завданні директив і машинних інструкцій повинна зустрічатись принаймні один раз.

Вимоги до машинних інструкцій тестової програми:

- повинен використовуватись кожен із режимів адресації, згідно із завданням;
- повинно бути забезпечено звернення до кожного із заданих типів (розрядностей) даних у пам’яті;
- повинен використовуватись хоча б один із регістрів даних заданої розрядності;
- повинна бути задана принаймні одна константа заданого формату (шістнадцяткова, десяткова, двійкова і т.п.);
- повинна бути задана принаймні одна константа (абсолютний вираз) заданої розрядності;
- команди передачі управління за умовою та внутрішньо-сегментні безумовні (внутрішньо-сегментні виклики процедур) повинні бути задані принаймні два рази – з посиланням вперед та посиланням назад.

Тестування пункту «*префікси заміни сегментів при необхідності автоматично генеруються транслятором*» повинно здійснюватись наступним чином: якщо у завданні вказано, що програма повинна містити лише один сегмент даних і один сегмент кодів, тоді у сегменті кодів необхідно задати директиву визначення пам’яті (DB, DW чи DD) з обов’язковою вказівкою імені та забезпечити звертання до цих даних принаймні в одній із машинних команд. Якщо програма може містити принаймні два логічних сегменти даних, тоді один із них у директиві ASSUME необхідно прив’язати до сегментного регістру, який не використовується за замовчуванням, та забезпечити відповідне звернення до даних із цього сегменту.

Для перевірки повноти тестової програми необхідно проаналізувати наявність перевірки кожного пункту завдання у тестовому файлі.

На першому етапі виконання роботи студенти повинні підготувати два тестові файли: файл тестової програми для свого компілятора та модифікований файл для MASM (TASM). Суть модифікації полягає у забезпеченні *режимів і умов для трансляції тестової програми* для компілятора MASM (TASM). До таких режимів і умов належать наступні:

1. При використанні у тестовій програмі *32-розрядних адрес і даних* першою директивою модифікованого тесту повинна бути директива *.386.*
2. Якщо у завданні існує пункт *«16-розрядні дані та зміщення* в сегменті, у випадку *32-розрядних даних та 32-розрядних зміщень* генеруються відповідні префікси зміни розрядності», то в директивах Segment модифікованого тесту необхідно задати операнд *Use16*.
3. Якщо в ідентифікаторах можуть використовуватись *букви українського алфавіту*, то у модифікованому файлі букви українського алфавіту необхідно замінити на букви латинського (транслітерація).
4. При відсутності у тестовій програмі 32-розрядних даних і 32-розрядних адрес, її необхідно без модифікації протранслювати за допомогою MASM (TASM). У разі виявлення помилок, пов’язаних з відсутністю відповідних команд у попередніх процесорах фірми Intel, потрібно модифікувати тестовий файл згідно з п.1 (.386) та п.2 (Use16). При відсутності помилок, тестові файли для MASM (TASM) і для компілятора студента можуть повністю співпадати.

Зразок тестового файла, що створений для **Прикладу індивідуального завдання 1** (див. п.2), представлено у Додатку В.

4. МЕТОДИКА РОЗРОБКИ КОМПІЛЯТОРА

4.1. Структурна схема компілятора Асемблера

Теорія та практика створення компіляторів мов програмування досить глибоко та всебічно розвинута та буде вивчатись у відповідній дисципліні. Побудова компіляторів мов високого рівня ґрунтується на використанні формальних граматик, теорії цифрових автоматів та ін. Більшість сучасних компіляторів Асемблера використовують досягнення у побудові компіляторів мов високого рівня. Але, на жаль, при цьому втрачається прозорість та логічна простота процесу компіляції програм, написаних мовою Асемблера. Використання фактично надлишкових можливостей у деякій мірі відвертає увагу від основоположних проблем, які вирішуються як при створенні мови Асемблера, так і при трансляції програм мовою Асемблера. Тому будемо розглядати побудову Асемблера за простою класичною двохпереглядною схемою з використанням прямих методів трансляції.

Перегляд – послідовна обробка символів файла початкової програми. Послідовність двох рівнів – послідовність рядків програми та послідовність символів у рядках. У випадку класичної двохпереглядної схеми Асемблера файл початкової програми переглядається 2 рази.

Прямий метод трансляції – почергове порівняння рядка програми або частини рядка з можливими синтаксичними конструкціями мови Асемблера. Звичайно, прямі методи трансляції доцільно використовувати

лише при незначній кількості можливих синтаксичних конструкцій, що притаманно мові Асемблера.

Структурна схема компілятора за класичною двохпереглядною схемою приведена на рис.4.1. Будемо вважати, що початкова програма представлена в деякому файлі у символьному форматі, наприклад, у коді ASCII з розширенням буквами українського алфавіту, тобто код кожного символу програми міститься в одному байті. Для спрощення будемо вважати, що будь-яка інструкція програми повністю розміщується в одному рядку. Компілятор на кожному перегляді послідовно читає рядки програми, а прочитавши рядок – послідовно переглядає символи рядка. Коли компілятор розпізнає директиву END, він закінчує роботу по черговому перегляду.

Розпізнавання з точки зору програми – це передавання управління на ту частину програми (або процедуру), де виконується обробка тієї чи іншої частини рядка. У загальному випадку, для такого передавання управління необхідно виконати досить складний аналіз як поточних символів рядка так і структур даних, які створені при обробці попередніх рядків або при створенні самого компілятора.

Задачі, які вирішуються на переглядах

Задачі, які вирішуються на ***першому перегляді***:

1. *Розподіл пам'яті*, тобто визначення зміщень команд та даних у відповідних логічних сегментах. Для вирішення цієї задачі мова Асемблера повинна давати можливість визначення кількості байтів, які будуть генеруватись за кожним рядком програми. Визначення їх вмісту на першому перегляді у класичній двохпереглядній схемі не передбачається.

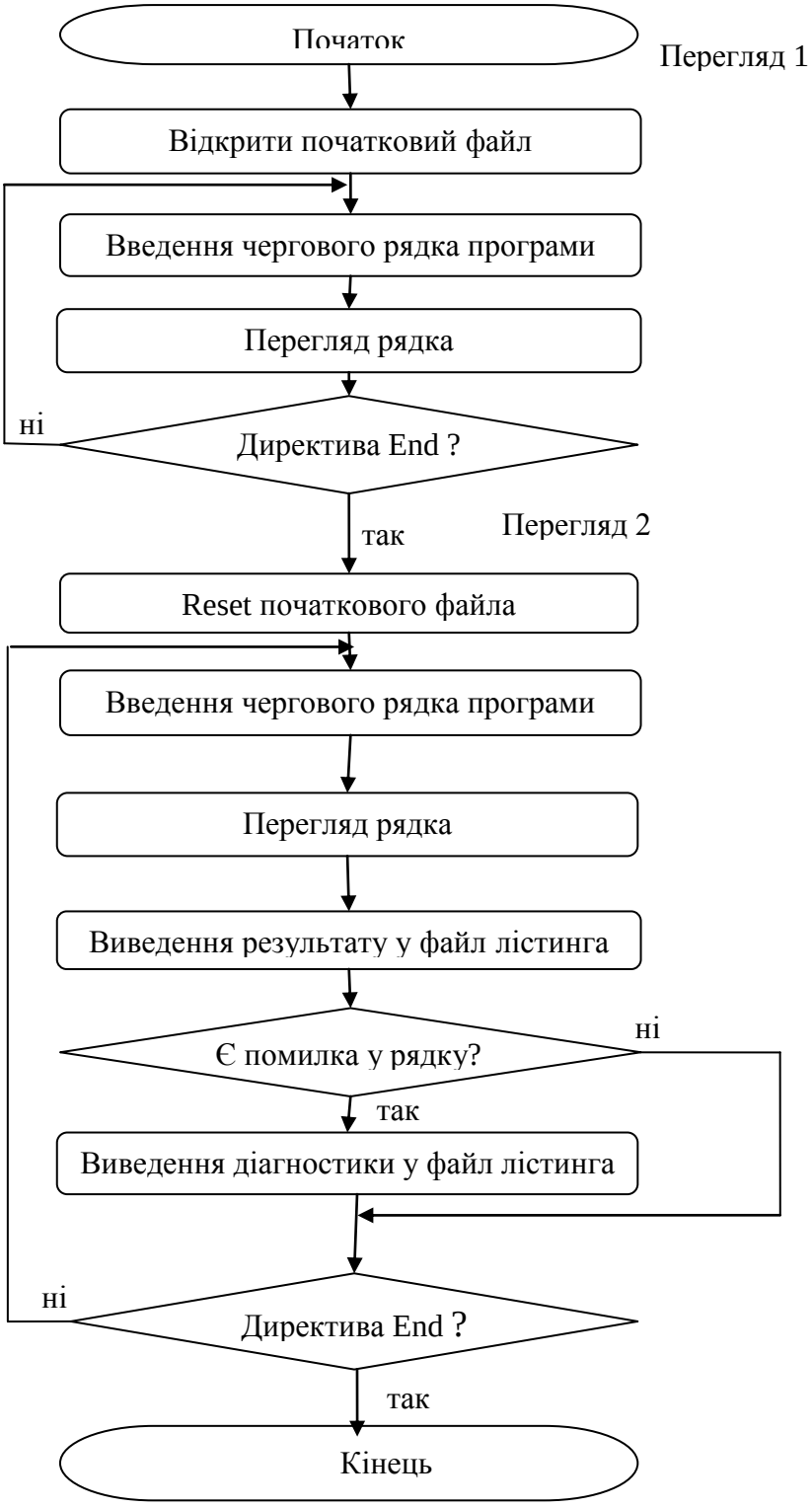


Рис.4.1. Структурна схема компілятора

2. *Формування таблиці ідентифікаторів*, які визначаються користувачем. До таких ідентифікаторів належать:
    - *мітки* (символічні адреси команд, тобто зміщення першого байту команди у відповідному логічному сегменті);
    - *імена у директивах визначення пам'яті* (символічні адреси даних – зміщення молодшого байту даних у відповідному логічному сегменті);
    - *імена процедур* (символічні адреси процедур – зміщення першого байту коду процедури у відповідному логічному сегменті);
    - *символічне позначення* відповідних послідовностей символів у директиві EQU;
    - *символічне позначення констант* у директиві = ;
    - *імена макрокоманд*;
    - *імена програмних об'єктів* у директиві Extrn.
3. *Виявлення помилки* – повторне визначення одного й того ж ідентифікатора.
4. *Формування таблиці логічних сегментів*.
5. *Формування бібліотеки макровизначень*.
6. *Формування макророзширень для макрокоманд*.

Задачі, які вирішуються ***на другому перегляді***:

1. *Генерування вмісту байтів команд та байтів даних* за рядками початкової програми.
2. *Формування файла лістинга* з додаванням таблиці логічних сегментів і таблиці ідентифікаторів, визначених користувачем (з першого перегляду).

3. *Формування повідомлень про помилки* у файлі лістинга, також про помилки повторного визначення ідентифікаторів, які виявлені на першому перегляді.

Дії та структури даних Асемблера на кожному з переглядів

Для вирішення вказаних задач як на першому, так і на другому перегляді над кожним рядком програми виконуються наступні дії:

- лексичний аналіз;
- *визначення структури речення програми* (визначення полів машинної інструкції або директиви) – елементи синтаксичного аналізу;
- елементи граматичного аналізу, в результаті якого безпосередньо виконуються задачі, що покладаються на відповідний перегляд.

Оскільки перелік дій компілятора для першого і другого перегляду співпадають, при розгляді відповідних дій увага буде акцентуватись на особливостях кожного з переглядів.

Під час роботи компілятора використовується та формується ряд таблиць:

1) *Таблиця зарезервованих ідентифікаторів* (ключових слів мови Асемблера), до яких належать мнемокоди машинних інструкцій і директив Асемблера, імена регістрів, імена операторів та операцій, імена операндів в окремих директивах та ін. Ця таблиця формується при розробці компілятора та є незмінною.
2) *Таблиця ідентифікаторів, які визначив користувач* у програмі (імена, мітки і т.п.). Ця таблиця формується на першому перегляді, на другому перегляді лише використовується і не доповнюється

(можлива модифікація у випадку визначення абсолютного значення директивою =).

3) *Таблиця логічних сегментів*. Ця таблиця формується на першому перегляді, та частково модифікується на другому перегляді.
4) *Таблиця назначень сегментних регістрів*. Ця таблиця заповнюється значенням NOTHING на початку переглядів та модифікується при обробці директиви Assume в обох переглядах.
5) *Таблиця лексем*. Ця таблиця формується для кожного рядка програми на обох переглядах та після обробки чергового рядка обнуляється.
6) *Таблиця структури рядка програми*. Ця таблиця формується для кожного рядка програми на обох переглядах та після обробки чергового рядка обнуляється.
7) *Бібліотека макровизначень*. Ця бібліотека формується на першому перегляді, а на другому перегляді лише використовується і не змінюється.
8) *Таблиця формальних параметрів макрокоманди*. Ця таблиця формується при обробці директиви Macro, а при обробці директиви Endm обнуляється.
9) *Таблиця фактичних параметрів макрокоманди*. Ця таблиця формується при створенні макророзширення, а при обробці директиви Endm обнуляється.

Із приведеного вище випливає, що при побудові компілятора Асемблера важливе значення має створення процедур читання та записування даних у таблиці.

4.2. Лексичний аналіз

В результаті лексичного аналізу формується *таблиця лексем* чергового рядка програми.

Лексема – один або більше символів у рядку, які Асемблером розглядаються як єдиний об'єкт. Лексеми – "неподільні атоми" мови програмування. У зв'язку з цим часто використовується термін «термінальний символ» мови програмування. В мові Асемблера розрізняють наступні лексеми:

- *Ідентифікатори* – послідовності букв та цифр, які розпочинаються з букви.

До букв відносяться малі та великі букви відповідних алфавітів. В MASM та TASM таким алфавітом є латинський алфавіт, а також символи _, @, ?, $. Загальноприйнято не розрізняти малі та великі букви. В той же час малі та великі букви в ASCII мають різні коди. Тому при обробці ідентифікаторів компілятор перетворює усі малі букви у великі. Якщо компілятор розробляється, наприклад, в Україні, то доречно в якості букв ідентифікаторів ввести і букви українського алфавіту. Латинський та український алфавіти мають багато однакових за зовнішнім виглядом букв, але вони мають різні коди ASCII (наприклад а, е, р, х та ін.). Для уникнення потенційних помилок, пов'язаних з неправильним переключенням алфавіту при введенні початкових програм, компілятор при обробці ідентифікаторів повинен для однакових за зовнішнім виглядом букв латинського та українського алфавітів вибрати один і той самий код. Довжина ідентифікатора практично не обмежується, але значущими є перші 32 символи.

- *Числові константи* – послідовності цифр та деяких букв, які розпочинаються з цифри.

Розрізняють двійкові, вісімкові, десяткові та шістнадцяткові константи. В якості допустимих букв числової константи є великі та малі (вони як і у ідентифікаторах не розрізняються) букви a, b, c, d, e, f та букви основи системи числення – b, o, q, d, h.

Двійкові константи складаються із цифр 0 та 1 і обов'язково закінчуються буквою b.

Вісімкові константи складаються із цифр від 0 до 7 і обов'язково закінчуються буквою o або q.

Десяткові константи складаються із цифр від 0 до 9 та можуть закінчуватись буквою d. Особливістю десяткових констант є те, що в них допускається відсутність букви d. Ознакою закінчення десяткової константи в цьому випадку є поява символу, який не входить до переліку цифр від 0 до 9.

Шістнадцяткові константи складаються із цифр від 0 до 9 і букв a, b, c, d, e, f та обов'язково закінчуються буквою h. Якщо першим символом шістнадцяткової константи є буква, то спершу необхідно поставити цифру 0, щоб компілятор зміг відрізнити ідентифікатор від числової константи.

- *Текстові константи* – послідовність довільних символів, які можна ввести з клавіатури та які розпочинаються і закінчуються символом ‘ або “. Символи у текстових константах ніяким перетворенням не підлягають.
- *Розподільники лексем* – символи пробілу, табуляції та крапка з комою.
- *Односимвольні лексеми* – всі інші символи, які входять до алфавіту мови Асемблера.

Розподільник між двома лексемами є обов'язковим, якщо конкатенація (при стикування) цих лексем є також лексемою.

Таблиця лексем може мати, наприклад, наступну структуру (табл. 4.1):

Таблиця 4.1

Структура таблиці лексем

№ п/п	Лексема	Довжина лексеми у символах	Тип лексеми

Наприклад, для рядка

*@30: mov eax, dword ptr gs:Aray1[ebx+esi*4+40h]; формування покажчика*

буде створена наступна таблиця із 19 лексем (табл. 4.2):

Таблиця 4.2

Приклад 1заповнення таблиці лексем

№ п/п	Лексема	Довжина лексеми у символах	Тип лексеми
1.	@30	3	Ідентифікатор користувача або не визначений
2.	:	1	односимвольна
3.	MOV	3	Ідентифікатор мнемокоду машинної інструкції
4.	EAX	3	Ідентифікатор 32- розрядного регістра даних
№ п/п	Лексема	Довжина лексеми в символах	Тип лексеми
5	,	1	односимвольна
6	DWORD	5	Ідентифікатор тип 4
7.	PTR	3	Ідентифікатор оператора визначення типу
8	GS	2	Ідентифікатор сегментного регістра
9	:	1	односимвольна

Продовження табл. 4.2

№ п/п	Лексема	Довжина лексеми у символах	Тип лексеми
10	ARAY1	5	Ідентифікатор користувача або не визначений
11	[	1	односимвольна
12	EBX	3	Ідентифікатор 32- розрядного регістра адрес
13	+	1	односимвольна
14	ESI		Ідентифікатор 32- розрядного регістра адрес
15	*	1	односимвольна
16	4	1	Десяткова константа
17	+	1	односимвольна
18	40H	3	Шістнадцяткова константа
19	]	1	односимвольна

Для рядка

My_data db 10100b, ’зв’язок’

буде створена інша таблиця лексем (табл. 4.3).

Таблиця 4.3

Приклад 2 заповнення таблиці лексем

№ п/п	Лексема	Довжина лексеми у символах	Тип лексеми
1	My_data	7	Ідентифікатор користувача або не визначений
2	DB	2	Ідентифікатор директиви даних тип 1
3	10100B	6	Двійкова константа
4	,	1	односимвольна
5	зв’язок	7	Текстова константа

В результаті формування таблиці лексем із подальшого розгляду вилучаються всі розподільники та коментарі.

Характерною помилкою, що виявляється при лексичному аналізі, є недопустимий символ, тобто поява у реченні за межами текстової константи символу, який можна ввести з клавіатури, але який не належить до алфавіту мови Асемблера.

Управління джерелом даних у лексичному аналізаторі

Як відмічалось раніше, компілятор програм мовою Асемблера здійснює послідовний аналіз рядків початкової програми. Лексичний аналізатор виконує посимвольний аналіз чергового рядка програми, тобто джерелом символів лексичного аналізатора є введений рядок програми. Але джерелом символів можуть бути визначення ідентифікаторів у директивах EQU, які беруться із таблиці ідентифікаторів користувача. Крім того, при формуванні макророзширень чергові рядки програми читаються не з початкового файла, а із бібліотеки макровизначень. У зв'язку з цим, в компіляторах рекомендується формувати окрему процедуру введення чергового символу програми, в якій виконуються необхідні переключення джерел вхідних даних. Це дає можливість концентрувати увагу як в лексичному аналізаторі, так і в інших модулях компілятора виключно на притаманних їм задачах.

4.3. Елементи синтаксичного аналізу

Визначення структури речення програми

Після лексичного аналізу перелік допустимих послідовностей лексем у можливій структурі речень в таблиці лексем може бути зведений до наступних:

<pusto>

<label>:

<mnem>

<mnem> {<op>}

<label>: <mnem>

<name> <mnem>

<label>: <mnem> {<op>}

<name> <mnem> {<op>}

де

<pusto> – таблиця, в якій відсутні лексеми (у випадку порожніх речень та речень коментарів);

<label> – ідентифікатор мітки в машинній інструкції;

<mnem> – ідентифікатор директиви, машинної інструкції чи макрокоманди або послідовність двох ідентифікаторів, одним із яких є префікс повторення, а другим – мнемоніка рядкової команди;

<name> – ідентифікатор імені в директиві;

{<op>} – один або більше операндів, які у більшості випадків є послідовністю лексем, розділених символом коми (кожна послідовність – окремий операнд або послідовність ідентифікаторів).

Відкидаючи випадок порожньої таблиці лексем, можна зробити наступні висновки щодо наведених послідовностей. По-перше, першою лексемою рядка завжди є ідентифікатор (якщо інша лексема – це вказує на синтаксичну помилку і подальший аналіз може бути припинено). По-друге, необхідно за допомогою процедури пошуку у таблицях перевірити наявність цього ідентифікатора серед мнемокодів машинних інструкцій, мнемокодів директив і мнемокодів макрокоманд. В результаті такого аналізу створюється таблиця структури речення, де вказуються порядкові номери лексем із таблиці лексем, які належать полю міток, полю мнемокоду та кожному із операндів поля операндів. Таблиця структури речення може мати наступний вигляд (табл. 4.4).

Таблиця 4.4

Таблиця структури речення

Поле міток (імені)	Поле мнемокоду		1-й операнд		2-й операнд		n-й операнд	
№ лексеми поля	№ першої лексеми поля	Кіл-ть лексем поля	№ першої лексеми операнда	Кіл-ть лексем операнда	№ першої лексеми операнда	Кіл-ть лексем операнда	№ першої лексеми операнда	Кіл-ть лексем операнда

Наявність у полі мнемокоду двох лексем відповідає випадкам використання префіксів повторення для ланцюгових (рядкових) команд.

Відповідно до двох попередніх прикладів рядків програми (табл. 4.2 і 4.3 п. 4.2), маємо (табл. 4.5):

Таблиця 4.5

Приклади структури речень

Поле міток (імені)	Поле мнемокоду		1-й операнд		2-й операнд	
№ лексеми поля	№ першої лексеми поля	Кількість лексем поля	№ першої лексеми операнда	Кількість лексем операнда	№ першої лексеми операнда	Кількість лексем операнда
1	3	1	4	1	6	14
1	2	1	3	1	5	1

Аналіз операндів машинних інструкцій

Машинні інструкції можуть мати наступні операнди:

- *константи* (абсолютні терми) або *вирази над константами* (абсолютні вирази) – використовуються для завдання безпосередніх даних;
- *регістри даних* – у більшості випадків використовуються для формування поля *reg байта modR/m*;
- *перемістимі вирази* – визначення складових зміщення (оператор *offset*) та імені логічного сегменту (оператор *seg*), які використовуються для завдання безпосередніх даних у командах. На відміну від абсолютних виразів, перемістимі вирази формують у командах безпосередні дані, які можуть змінюватися при формуванні виконавчих файлів та при завантаженні програм у пам’ять;
- *адресні вирази* використовуються для завдання адресних частин команд – байтів режиму адресації та зміщення в команді, (як виняток, в командах міжсегментної передачі управління – для завдання логічної адреси). Адресні вирази розпізнаються при наявності в операнді адресних термів (міток або імен, які визначають три складові: ім’я логічного сегменту, зміщення в логічному сегменті та тип) і/або при наявності адресних регістрів. У структурі адресних виразів допускається наявність абсолютних виразів.

Шляхом послідовного аналізу лексем, які входять в операнд, заповнюється таблиця лексем операндів (табл. 4.6). Використовуючи цю таблицю та таблицю структури речення (табл. 4.4), легко визначити, чим є операнд – регістром даних, перемістимим виразом, абсолютним виразом чи адресним виразом. Деякі ускладнення можуть виникнути лише у випадку, коли в абсолютному виразі використовуються перемістимі терми.

Таблиця 4.6

Лексеми операнда

№ п/п	Перелік можливих лексем операнда	Основні властивості	Допоміжні властивості
1	Наявність регістра даних	Розрядність	Номер регістра даних
2	Наявність оператора ptr	Тип	
3	Наявність префікса заміни сегмента	Номер сегментного регістра	
4	Наявність ідентифікатора мітки або імені	Номер рядка в таблиці ідентифікаторів користувача	
5	Наявність невизначеного ідентифікатора		
6	Наявність адресного регістра 1	Номер регістра адрес	Розрядність
7	Наявність адресного регістру 2		Розрядність
8	Наявність множника	Номер регістра адрес, який має множник	Значення множника
9	Наявність операторів offset або seg		
10	Наявність констант (абсолютного виразу)		

ЗАУВАЖЕННЯ. Номери регістрів процесора, що визначені фірмою Intel, представлені у Додатку Б.

Для машинної інструкції

*@30: mov eax,dword ptr gs:Aray1[ebx+esi*4+40h]; формування покажчика*

згідно табл. 4.5 та табл. 4.6 для першого операнда маємо (табл. 4.7):

Таблиця 4.7

Приклад таблиці лексем для першого операнда

N п/п	Перелік можливих лексем операнда	Основні властивості	Допоміжні властивості
1	True	32	0
2	False	-	
3	False	-	
4	False	-	
5	False	-	
6	False	-	
7	False	-	
8	False	-	
9	False	-	
10	False	-	

Із табл. 4.7 випливає, що операнд – 32-розрядний регістр даних, при цьому наявність будь-яких інших лексем в операнді розцінюється як помилка. Для другого операнда маємо (табл. 4.8):

Таблиця 4.8

Приклад таблиці лексем для другого операнда

N№ п/п	Перелік можливих лексем операнда	Основні властивості	Допоміжні властивості
1	False	-	-
2	True	4	
3	True	65h	
4	False	-	
5	True		
6	True	3	32
7	True	6	32
8	True	6	4
9	False		
10	True	40h	

Обробка абсолютних виразів

Обробка абсолютних виразів компілятором Асемблера полягає в обчисленні значення виразу. У загальному випадку існує значна кількість різних абсолютних виразів, тому використовувати прямі методи трансляції (шляхом перегляду всіх варіантів) практично неможливо. Для обчислення абсолютних виразів використовується алгоритм Дейкстри – використання стеку з пріоритетами. Суть алгоритму полягає у наступному:

1. Якщо чергова лексема виразу є абсолютним термом, його значення безумовно записується в стек даних.
2. Якщо чергова лексема виразу є операцією – порівнюється пріоритет операції на верхівці стеку та пріоритет поточної операції.
3. Якщо пріоритет поточної операції вище пріоритету операції на верхівці стеку, поточна операція записується в стек операцій. Перейти до п. 1.
4. Якщо пріоритет поточної операції менший або дорівнює пріоритету операції на верхівці стеку, операція читається із стеку операцій. Визначається кількість операндів цієї операції. Відповідна кількість даних читається із стеку даних. Операція, що прочитана із стеку операцій, виконується над прочитаними із стеку даними, а результат записується в стек даних. Перейти до п.3.
5. Якщо чергова лексема є відкриваючою круглою дужкою, вона безумовно записується в стек операцій. Вважається, що відкриваюча дужка має найнижчий пріоритет. Перейти до п. 1.
6. Якщо чергова лексема є закриваючою круглою дужкою, вона почергово виштовхує із стеку всі операції до відкриваючої круглої дужки. Потім відкриваюча та закриваюча круглі дужки знищуються. Перейти до п. 1.

7. Якщо всі лексеми вичерпані, а стеки не порожні, почергово виштовхуються із стеку всі операції.

Розглянемо, наприклад, обчислення абсолютного виразу 34+50*(7+10) (рис. 4.2):

№ кроку (поточна лексема)	Стек даних	Стек операцій
1. 34	34	
2. +	34	+
3. 50	50 34	+
4. *	50 34	* +
5. (	50 34	(* +
6. 7	7 50 34	(* +
7. +	7 50 34	+ (* +

№ кроку (поточна лексема)	Стек даних	Стек операцій
8. 10	10	+
	7	(
	50	*
	34	+
9. виштовхується +	17	(
	50	*
	34	+
10. виштовхується (	17	*
	50	+
	34	
11. кінець операнда виштовхується *	850	+
	34	
12. кінець операнда виштовхується +	884	

Рис. 4.2. Обчислення виразу 34+50*(7+10)

Наприкінці обчислення абсолютного виразу визначається кількість байтів, які необхідні для збереження значення виразу. Для розглянутого прикладу вона дорівнює 2.

4.4. Елементи граматичного аналізу

Після створення таблиці структури речення за полем мнемокоду, шляхом використання процедури пошуку в таблиці ключових слів мови Асемблера визначається вид речення: машинна інструкція чи директива. Виклад аналізу речень програми доцільно проводити у наступному порядку.

Аналіз основних директив програми на першому та другому переглядах

Обробка директив Segment та Ends

Формування (активізація) чергового рядка таблиці сегментів (див. табл. 4.9).

Таблиця 4.9

Таблиця сегментів

Ім’я сегмента	Розрядність за замовчуванням	Поточне зміщення	Розміщення	Об’єднання	Клас
Code	32	0			

Active_seg = номер відповідного рядка таблиці сегментів

На початку перегляду *Active_seg=0*. Будемо вважати, що рядки таблиці сегментів нумеруються з 1.

Поточне зміщення часто називають лічильником адреси Асемблера у поточному логічному сегменті. Мовою Асемблера він позначається символом *$*. При відкритті логічного сегмента на обох переглядах *$=0*. На обох переглядах відбувається пошук ідентифікатора імені сегмента у таблиці сегментів. Якщо на першому перегляді пошук не вдалий, тоді створюється новий рядок у таблиці сегментів з заповненням відповідних полів таблиці за замовчуванням або відповідно до ідентифікаторів

операндів. Наступні директиви *Segment* з тим самим ім'ям сегмента лише встановлюють значення *Active_seg*.

Обробка директиви Ends полягає в наступному: *Active_seg = 0*.

ЗАУВАЖЕННЯ. Лише директиви *Assume*, *Equ =*, *Macro*, *Endm*, *Public*, *Extrn* та рядки-коментарі можуть розміщуватися за межами логічного сегмента. У всіх інших рядках програми наявність *Active_seg = 0* розглядається як помилка.

Обробка директив Assume

На початку кожного перегляду в кожен рядок таблиці назначень сегментним регістрам заноситься ключове слово *Nothing*. Обробка операндів директиви *Assume* полягає у записі в цю таблицю ідентифікатора, який міститься за символом *:* (двокрапка). Розглянемо приклад. На початку кожного з переглядів формується наступна таблиця (табл. 4.10).

Таблиця 4.10

Таблиця назначень сегментним регістрам

Сегментний регістр	Назначення
CS	Nothing
DS	Nothing
SS	Nothing
ES	Nothing
GS	Nothing
FS	Nothing

Після обробки директиви

Assume CS:code, DS:date

таблиця назначень матиме вигляд (табл. 4.11). А якщо далі в програмі з'явиться директива

Assume CS:code, DS:date2, gs:segs

таблиця матиме вигляд (табл. 4.12).

Таблиця 4.11

Назначення після обробки

Сегментний регістр	Назначення
CS	CODE
DS	DATE
SS	Nothing
ES	Nothing
GS	Nothing
FS	Nothing

Таблиця 4.12

Нове назначення

Сегментний регістр	Назначення
CS	CODE
DS	DATE2
SS	Nothing
ES	Nothing
GS	SEGS
FS	Nothing

Обробка директиви ORG <абсолютний вираз>

У рядку *Active_seg* таблиці сегментів у поле "*Поточне зміщення*" записується значення абсолютного виразу, тобто *$ = <абсолютний вираз>*. Значення *Active_seg = 0* означає розміщення директиви *ORG* поза логічним сегментом, що є помилкою.

Обробка поля міток машинних інструкцій і поля імені директив визначення пам'яті

Якщо поле імені (мітки) не порожнє, воно обробляється на першому перегляді за наступним алгоритмом (далі Алгоритм 1):

1) За допомогою процедури пошуку в таблицях виконати пошук ідентифікатора імені (мітки) в таблиці ідентифікаторів, визначених користувачем. При успіху пошуку перейти до п.3.
2) Записати в таблицю ідентифікаторів, визначених користувачем, мітку (ім'я), задавши значення зміщення із поля "*Поточне зміщення*" активного логічного сегмента значення сегментної складової – ім'я активного логічного сегмента (або номер рядка активного логічного сегмента в таблиці сегментів *Active_seg*), значення типу near (*-1*) для мітки або *1, 2, 4, 6, 8, 10* для директив *db, dw, dd, dp, dq, dt* відповідно. Перейти до п.4.
3) Зафіксувати помилку – дублювання імені (мітки).
4) Кінець.

На другому перегляді виконується пошук ідентифікатора у полі мітки (імені) таблиці ідентифікаторів користувача та порівнюється значення зміщення із значенням поля поточного зміщення активного сегмента у таблиці сегментів. При неспівпадінні значень формується діагностичне повідомлення "***Phase error"*** – не відповідність зміщень першого та другого переглядів.

Надалі ідентифікатор користувача, якому в таблиці назначено зміщення, тип та ім'я сегмента будемо називати *адресним термом*.

Обробка інших директив

Обробка директиви Proc та Endp

Ім'я процедури у полі імені обробляється згідно з вищезазначеним Алгоритмом 1. Значення типу визначається ключовим словом у полі операндів директиви: *-1* (якщо *near* або поле операндів порожнє), *-2* (якщо *far*). Директива *Endp* на першому перегляді не обробляється. На другому перегляді значення типу процедури записується у глобальну змінну

компілятора, яка використовується для генерації відповідного коду команди *Ret* (*Retn* чи *Retf*). Директива *Endp* на другому перегляді встановлює цю змінну в значення *-1* (*near*). Початкове значення цієї змінної також *-1*. Директива *Endp* на першому перегляді не оброляється.

Обробка директиви Equ

Ім'я у полі імені директиви обробляється згідно з Алгоритмом 1 обробки поля міток і лише на першому перегляді. При цьому в якості значення записується послідовність символів із поля операндів, а в якості типу – тип *Equ* (наприклад, значення *-3*). На другому перегляді ця директива не обробляється.

При лексичному аналізі необхідно відслідковувати появу визначеного ідентифікатора з типом *-3* і виконувати переключення процедури введення чергового символу програми на введення символів з поля значень такого ідентифікатора у таблиці ідентифікаторів користувача.

Обробка директиви =

Алгоритм 2(для обох переглядів):

1) Обчислити абсолютний вираз поля операндів директиви згідно з алгоритмом Дейкстри (див. стор. 30 *Обробка абсолютних виразів*).
2) Провести пошук ідентифікатора з поля імені директиви в таблиці ідентифікаторів користувача. При успіху пошуку – перейти до п.4.
3) Записати новий ідентифікатор в таблицю, задавши значення зміщення як значення абсолютного виразу поля операндів та значення типу – *abs* (наприклад, значення *0*). Перейти до п.5.
4) Записати нове значення зміщення у рядок таблиці знайденого ідентифікатора.
5) Кінець.

Обробка директиви Label

Виконується Алгоритм 1, в якому значення типу визначається за ключовим словом у полі операндів директиви.

Визначення кількості байтів, які генеруються за рядками програми

Після обробки поля імені (мітки) визначається кількість байтів, які Асемблер повинен генерувати за поточним реченням. Детальний алгоритм визначення кількості байтів розглянемо далі. Тут відмітимо, що визначене значення *k* кількості байтів додається до значення поля "*Поточне зміщення*" активного сегмента у таблиці сегментів (тобто *$ = $+ k*).

Розглянутий процес формування таблиці ідентифікаторів, визначених користувачем, та визначення кількості байтів за реченням називають *процесом автоматичного розподілу пам'яті програми* або просто *розподілом пам'яті*.

Визначення кількості байтів даних у директивах db, dw, dd, dp, dq, dt

На обох переглядах значення кількості байтів *k*, які необхідно згенерувати за відповідною директивою, визначається за формулою

$$k = type*count$$

де *type* (тип) визначається директивою, а *count* (у випадку відсутності операндів з повтореннями) – це кількість операндів директиви згідно з таблицею структури речення (табл. 4.4).

У випадку застосування операнда з оператором повторення *dup*, *count* – кількість повторень. Якщо ж у директиві використовуються операнди з вкладеними повтореннями або операнди як з повтореннями, так і без них, тоді для визначення *count* рекомендується використовувати рекурсивну процедуру. Якщо у директиві *db* використовується текстова константа, то *count* – це кількість символів відповідної лексеми з таблиці лексем.

На другому перегляді додатково необхідно згенерувати байти даних, попередньо обчисливши значення констант чи абсолютних виразів. При генеруванні байтів необхідно пам’ятати, що в процесорах фірми Intel молодші байти розміщуються за молодшими адресами. Деякі особливості має генерування, якщо в директивах *dw, dd, dp* в якості операнда задається мітка або символічне ім’я. Процес залежить від розрядності, яка задана в поточному сегменті (де знаходяться директиви *dw, dd, dp*) та розрядності сегмента, де знаходиться ім’я (мітка), і, відповідно, розрядності зміщення імені (мітки).

Правила відповідають здоровому глузду. В директиві *dw* може використовуватись лише 16-розрядне зміщення у 16-розрядному сегменті, при цьому із таблиці ідентифікаторів користувача читається відповідне 16-розрядне зміщення і на його основі генеруються байти програми. У директиві *dd* в аналогічному випадку додатково генеруються ще два старших нульових байти для значення сегментної складової, яку визначить завантажувач. У директиві *dd* у випадку 32-розрядного зміщення та 32 розрядного поточного сегмента генеруються 4 байти цього зміщення. У такому ж випадку, для директиви *dp* додатково генеруються ще два старших нульових байти для значення сегментної складової. Всі інші випадки помилкові (хоча *Masm* та і *Tasm* не завжди вказують на помилку).

Визначення кількості байтів машинної інструкції (на обох переглядах) та генерування команд процесора на другому перегляді

На обох переглядах обробка адресних виразів використовується для визначення кількості байтів, які необхідно згенерувати за поточною машинною інструкцією.

Нагадаємо структуру коду команди процесорів 80x86 (рис. 4.3).

1, 2 або 3 однобайтні префікси	1 або 2 байти коду операції	0, 1 або 2 байти режиму адресації (ModR/m та Sib)	0, 1, 2 або 4 байти зміщення в команді	0, 1, 2 або 4 байти безпосередніх даних

Рис. 4.3. Структура коду команди процесорів 80x86

Розглянемо окремо кожне поле коду команди.

Код операції

Кількість байтів коду операції однозначно визначається мнемонікою команди і записується в один із стовпців таблиці ключових слів. У деяких командах частина коду операції розташовується в полі *reg* байта *modR/m*. У цьому випадку, у відповідний рядок таблиці ключових слів записується, що код операції є однобайтним, та ознака безумовної присутності байта *modR/m*. В результаті, значення кількості байтів з таблиці ключових слів заноситься як початкове значення кількості байтів *k*, які формуються на основі поточної інструкції.

В той же час, значення байту (байтів) коду операції (на другому перегляді) далеко не завжди визначається мнемонікою. Необхідний додатковий аналіз поля операндів. При цьому існують мнемоніки, коли коди операції кардинально залежать від поля операндів. Це дещо ускладнює структуру рядків таблиці ключових слів, які презентують мнемоніки машинних інструкцій, оскільки необхідно заносити в рядок усі можливі заготовки для кодів операцій. Наприклад, значна кількість команд в 0-му біті коду операції містить *ознаку розрядності даних 0* – для байтів, *1* – для слів або подвійних слів (конкретне значення – за замовчуванням або змінюється при наявності префікса заміни розрядності даних – див.

наступні пункти). Крім того, для багатьох команд *1-й* біт коду операції вказує, який саме операнд є приймачем: *0* – приймач визначається полем *r/m, 1* - полем *reg*.

Визначення розрядності зміщення в команді, наявність префікса заміни розрядності адрес

Поле зміщення присутнє в команді, якщо один з операндів є адресним виразом, який, в свою чергу, містить визначений або невизначений ідентифікатор користувача, абсолютний вираз чи константу. Всі необхідні дані містяться в таблицях згідно п.4.2. Із цих же таблиць, у випадку наявності регістрів адрес (регістри у квадратних дужках), визначається можлива розрядність поля зміщення в команді – *1* або *2* байти при 16-розрядних адресних регістрах, *1* або *4* байти при 32-розрядних.

Якщо адресний вираз містить адресний терм або невизначений ідентифікатор, кількість байтів зміщення в команді може дорівнювати *2* або *4*, навіть у випадку, коли зміщення в сегменті для такого ідентифікатора є однобайтним. Це пояснюється тим, що однобайтне зміщення в сегменті під час редагування зв'язків може стати двох- або чотирьохбайтним. Але при роботі редактора зв'язків вставити у програму додатковий байт практично неможливо. Якщо ж в адресному виразі немає адресного терму або невизначеного ідентифікатора, розрядність поля зміщення в команді визначається значенням абсолютного виразу чи константи: *1* байт (якщо значення більше або дорівнює -128 і менше 128), *2* або *4* байти (в інших випадках).

Визначена кількість байтів зміщення в команді (*0, 1, 2* або *4*) додається до *k*.

На другому перегляді до зміщення адресного терму (якщо він є) додається значення абсолютного виразу (якщо він є) та за отриманим результатом генеруються відповідні байти поля зміщення в команді.

Якщо розрядність регістрів (регістра) адрес відрізняється від значення в полі розрядності активного сегмента, тоді встановлюється *k= k+1*, а на другому перегляді додатково генерується *префікс заміни розрядності адрес* (код – 67h).

Розрядність даних, наявність префіксів заміни розрядності даних

Розрядність даних у машинній інструкції визначається розрядністю регістра даних або оператором *ptr*, або типом адресного терму.

Якщо в операндах машинної інструкції існує більш ніж один із перелічених варіантів, то всі вони повинні вказувати на одну й ту саму розрядність. Якщо в операндах машинної інструкції немає жодного з перелічених варіантів, це свідчить про помилку в інструкції. Якщо визначена розрядність 8, префікс заміни розрядності даних не генерується. Якщо визначена розрядність даних 16 або 32, а в полі розрядності активного сегмента вказана інша розрядність, тоді установлюється *КБ = КБ+1*, а на другому перегляді додатково генерується *префікс заміни розрядності даних* (код – 66h).

Якщо в машинній інструкції є операнд, який є абсолютним виразом або константою, кількість байтів безпосередніх даних відповідає визначеній вище розрядності даних. У ряді команд допускається використання однобайтних безпосередніх даних при розрядності даних у два або 4 байти та значенні абсолютного виразу в межах від -128 до 127. При цьому відповідний біт коду операції встановлюється в 1, а при виконанні команди процесор виконує знакове розширення однобайтних безпосередніх даних.

Байт префіксу заміни сегмента

Алгоритм визначення байта не явно заданого префікса заміни сегмента полягає у наступному:

1) Визначається сегментний регістр за замовчуванням (за переліком регістрів адрес). Нагадаємо, що, як правило, за замовчуванням використовується регістр *DS*. Виключення наступні: у випадку використання *BP* як адресного регістра або регістрів *EBP* і *ESP* (без множника) як базових, за замовчуванням використовується регістр *SS*.
2) Якщо операнди машинної інструкції не є адресними виразами, або в адресних виразах відсутній адресний терм, або це команда передачі управління з типом операнда *Far* (*Near*), байт заміни сегмента відсутній.
3) За таблицею ідентифікаторів користувача визначається ім'я логічного сегмента, в якому розміщений адресний терм.
4) У таблиці назначень сегментним регістрам (див. Обробку директиви Assume) визначається сегментний регістр, якому назначено логічний сегмент, де саме розміщується ідентифікатор.
5) Якщо логічному сегменту назначено сегментний регістр, який не співпадає з сегментним регістром за замовчуванням, тоді повинен генеруватись байт префікса заміни сегмента назначеного сегментного регістра та встановлюватись $k = k+1$ (значення байту заміни сегмента для генерації на другому перегляді згідно з Додатком Б).

При явному завданні префікса заміни сегмента визначається сегментний регістр за замовчуванням (згідно з вищезазначеним), і, при

його співпадінні з явно заданим сегментним регістром, префікс не генерується.

Байти режимів адресації

На обох переглядах необхідно визначити наявність та кількість байтів (один чи два) режимів адресації. На другому перегляді додатково визначаються значення цих байтів.

Можна виділить наступні групи машинних інструкцій:

- перша група – поле режиму адресації для заданої мнемоніки завжди відсутнє;
- друга група – поле режиму адресації для заданої мнемоніки завжди присутнє;
- третя група – поле режиму адресації може бути та залежить від поля операндів;
- четверта група – одній і тій самій машинній інструкції мовою Асемблера можуть відповідати дві машинні команди з абсолютно однаковими діями, але з байтами режимів адресації або без них.

До першої групи відносяться, наприклад, команди управління ознаками (*Cli, Sti, Clc, Stc, Cmc, Cld, Std*), команди корекції для двійково-десяткових обчислень (*Aaa, Aas, Daa, Das*), команди передачі управління за умовою, команди обробки масивів – рядкові (ланцюгові) команди, команди перетворення типів (байт-слово, слово-подвійне слово) та ін.

До другої групи належать, наприклад, команди *Lea, Movsx, Movzx, Neg, Not*, команди обробки бітових полів, команди множення та ділення без знакових чисел, команди завантаження логічної адреси (*Lds, Lss, Les, Lgs, Lfs*), команди лінійних, арифметичних та циклічних зсувів.

До третьої групи відносяться команди *Jmp* та *Call*. При використанні прямої (відносної) адресації поле режиму адресації відсутнє. Якщо використовується опосередкована (непряма), тоді присутнє поле режиму адресації.

До четвертої групи відносяться команди традиційної обробки: пересилання, додавання, віднімання, порівняння, порозрядної логічної обробки. Оскільки ці команди найчастіше застосовуються в програмах, для частини з них, поряд із загальними реалізовані спрощені формати структури команди без поля режимів адресації. Наприклад, для команд, де першим з операндів є регістр *EAX* (*AX, AL*), а другим – безпосередні дані та ін. Це потребує більш детального аналізу команд вказаної групи для визначення наявності поля режиму адресації.

Поле режиму адресації складається із байта *Modr/m* та, можливо, байта *sib*. Наявність байту *sib* визначається адресним виразом при виконанні однієї із умов: наявності двох 32-розрядних адресних регістрів або наявності множника.

Формування байтів поля режимів адресації на другому перегляді

Структура байта *Modr/m* при 16-розрядній адресації має наступний вигляд (рис. 4.5):

Поле mod – 2 біти	Поле reg – 3 біти	Поле r/m – 3 біти

Рис. 4.5. Структура байта Modr/m при 16-розрядній адресації

Поле *mod* використовується для визначення зміщення в команді, а поле *r/m* – для визначення регістрів адрес, вміст яких використовується для

формування ефективної адреси. Поле *reg* призначене для завдання регістра даних, який використовується в команді або є частиною коду операції.

Розглянемо поле mod:

- При *mod=00* зміщення в команді відсутнє, а ефективна адреса формується за вмістом регістра (регістрів) адрес, які задаються полем *r/m*.
- При *mod=01* зміщення в команді однобайтне, яке при формуванні ефективної адреси знаково розширюється до двох байтів з подальшим додаванням до вмісту регістра (регістрів) адрес, які задаються полем *r/m*.
- При *mod=10* зміщення в команді двохбайтне і при формуванні ефективної адреси додається до вмісту регістра (регістрів) адрес, які задаються полем *r/m*.
- При *mod=11* адреса пам'яті не задається, а поле *r/m* задає код регістра даних.

У полі *r/m* регістри адрес або їх можлива комбінація задається наступним чином (табл.. 4.13):

Таблиця 4.13

Поле r/m у режимі 16-розрядної адресації

Код у полі r/m	Регістри адрес	Сегментний регістр, який використовується за замовчуванням
000	BX+SI	DS
001	BX+DI	DS
010	BP+SI	SS
011	BP+DI	SS
100	SI	DS
101	DI	DS
110	BP	SS
111	BX	DS

З табл. 4.13 можна зробити наступні висновки, які справедливі і для сучасних мікропроцесорів сімейства при 16-розрядній адресації:

1. При 16-розрядній адресації тільки чотири регістра – *BX, BP, SI* та *DI* – можуть використовуватися як регістри адрес.
2. Для формування багатокомпонентної адреси використовується тільки обмежений набір пар регістрів: (*BX, SI*), (*BX, DI*), (*BP, SI*) (*BP, DI*).
3. Із реалізації випадає один із широко вживаних режимів – режим прямої адресації, тобто режим, коли зміщення в команді і є зміщенням у сегменті.

Відносно останнього пункту інженери фірми Intel вимушені були прийняти наступне вирішення: ввести режим прямої адресації при *mod=00* та *r/m=110*, тобто, не дивлячись на *mod=00*, зміщення в команді задавати двохбайтним, якщо *r/m=110*. При цьому регістр *BP* не використовується. Це дуже нагадує "латку" в програмах. Але ця "латка" досить продумана. З апаратної реалізації випадає режим посередньої регістрової адресації з використанням регістра *BP* як регістра адреси, але використання цього режиму при стратегічному призначенні регістра *BP* як базового регістра структур даних стеку, малоймовірне. У крайньому випадку, можна використати режим при *mod=01* та нульовому байті зміщення в команді, що і реалізовано компіляторами програм мови Асемблера.

Використання засобів формування 32-розрядних адрес у реальному режимі

Свого часу при створенні процесора 80386 перед інженерами фірми Intel постала непроста проблема: з одного боку, необхідно переходити на 32-розрядні дані та адреси, щоб забезпечити конкурентоздатність нового процесора, а з іншого боку, необхідно забезпечити програмну сумісність

нового процесора зі старими. Очевидне вирішення проблеми – створення додаткових кодів операцій – було б не оптимальним. Дійсно, при цьому кількість кодів операцій необхідно було б збільшити у декілька разів (поряд з існуючими командами обробки 16- та 8-розрядних даних при 16-розрядних адресах, необхідно було б створити команди обробки 8, 16 та 32-розрядних даних при 32-розрядних адресах та команди обробки 32-розрядних даних при 16-розрядних адресах). Ця проблема була вирішена наступним чином.

По перше, встановлювалось глобальне (для програми в цілому) значення розрядності адрес та даних за замовчуванням: або 16, або 32. Наприклад, у реальному режимі, за замовчуванням, для всіх програм встановлювались 16-розрядні адреси та 8- або 16-розрядні дані, що властиве старим процесорам.

По друге, створювались однобайтні префікси команд для зміни розрядності, яка задана за замовчуванням (*66h – зміна глобального значення розрядності даних*, *67h – зміна глобального значення розрядності команд*). Дія цих префіксів обмежувалась рамками однієї команди, яка розташовувалась за префіксами. В реальному режимі наявність префіксу 66h забезпечувала при одному й тому ж коді операції використання 32-розрядних даних замість 16-розрядних. Якщо код операції вказував на 8-розрядні дані, тоді цей префікс на таку команду не впливав (та і необхідність префіксу 66h перед командами обробки 8-розрядних даних відпадає). Наявність префіксу 67h дозволяє використовувати більш гнучкий механізм 32-розрядної адресації.

Таким чином, у реальному режимі забезпечувалось виконання усіх програм, розроблених для попередніх процесорів. Щодо нових програм, то

вони могли обробляти 32-розрядні дані та використовувати механізм формування 32-розрядних адрес без створення додаткових кодів операцій.

В той же час, при модифікації мови Асемблера засоби явного завдання префіксів зміни розрядності не вводились. Формування префіксів зміни розрядності повністю покладалось на компілятор.

Інтерпретація полів байта *mod-r/m* у режимі 32-розрядної адресації відрізняється від режиму 16-розрядної адресації. Головна відмінність – в інтерпретації процесором поля *r/m* (табл.. 4.14):

Таблиця 4.14

Поле r/m у режимі 32-розрядної адресації

Код у полі r/m	Регістри адрес	Сегментний регістр, який використовується за замовчуванням
000	EAX	DS
001	ECX	DS
010	EDX	DS
011	EBX	DS
100	Наявність байта *Sib*	--
101	EBP*	SS
110	ESI	DS
111	EDI	DS

Байт *SIB* має наступну структуру (рис. 4.6):

Поле множника – 2 біти	Поле індексного регістра - 3 біти	Поле базового регістра - 3 біти

Рис. 4.6. Структура байта sib

У полі індексного регістра може вказуватись будь-який регістр за виключенням *ESP*. При формуванні ефективної адреси вміст регістра

зсувається вліво на кількість розрядів, які вказані у полі множника. Тим самим фактично при формуванні ефективної адреси відбувається множення вмісту індексного регістру на 2, 4 або 8, що позбавляє програміста додаткових дій при адресації елементів масивів слів, подвійних слів та квадрослів.

У полі базового регістра може використовуватися будь-який регістр загального призначення (*РЗП*), в тому числі і *ESP*.

Таким чином, можливість формування ефективної адреси у 32-розрядному режимі значно ширша, порівняно з 16-розрядним режимом адрес. Тому 32-розрядний режим за допомогою префікса зміни розрядності адрес може використовуватися і в реальному режимі при наступному уточненні – старші 16 розрядів ефективної адреси у реальному режимі ігноруються.

Окрім іншої інтерпретації поля *r/m*, по-іншому інтерпретується поле *mod*:

- при *mod=01* зміщення в команді однобайтне, яке при формуванні ефективної адреси знаково розширюється до чотирьох байтів з наступним додаванням до вмісту регістра (регістрів) адрес, які задаються полем *r/m* та *sib*;
- при *mod=10* зміщення в команді чотирьохбайтне і при формуванні ефективної адреси додається до вмісту регістра (регістрів) адрес, які задаються полем *r/m* та *sib*.

Приклади адресації у 32-розрядному режимі:

```
Mov  ax,Value[ecx*4+edx]
Mov  edx,[edx*8]
Mov  al,[eax+esp]
```

Як і у випадку 16-розрядних адрес, режим використання регістра *EBP* при *mod=0* відсутній. При коді *101* у полі *r/m* або у полі базового регістра байта *sib* встановлюється режим використання 4-х байтного зміщення у команді.

Особливості трансляції окремих команд

Команди передачі управління

Команди передачі управління, з точки зору мови Асемблера, поділяються на:

- команди з прямою адресацією (прямі), коли в операнді команди вказується мітка або ім’я процедури;
- опосередковані (непрямі), коли в операнді команди вказується адреса даних, які містять адресу переходу.

Трансляція непрямих команд передачі управління мало чим відрізняється від раніше викладеного, тому їх окремо розглядати немає сенсу.

Команди передачі управління розподіляються також на внутрішньосегментні та міжсегментні. Внутрішньосегментні прямі команди передачі управління з точки зору процесора мають відносну адресацію, коли цільова адреса формується як алгебраїчна сума вмісту (E)IP та зміщення в команді.

Команди передачі управління за умовою

Команди передачі управління за умовою можуть мати одну з наступних структур (рис. 4.7).

Один байт коду операції	Один байт зміщення у команді
Два байти коду операції	Два байти зміщення у команді

Рис. 4.7. Структура команди передачі управління за умовою

Визначається значення зміщення адресного терму в полі операндів.

На першому перегляді алгоритм обробки команди передачі управління за умовою наступний:

1) Визначається наявність мітки в полі операндів. Якщо в таблиці ідентифікаторів користувача відповідна мітка відсутня, тоді перейти до п.5).
2) Читається значення мітки з таблиці, і, якщо операнд додатково має абсолютний вираз, він обчислюється і отримане значення додається до значення зміщення мітки.
3) Обчислюється різниця між отриманим значенням та значенням поля «*Поточне зміщення*» відкритого логічного сегмента, збільшеним на довжину команди (*$+2*).
4) Якщо різниця менша за -128 або більша за 127, потрібно перейти до п.5).
5) Прийняти *k=2*. Перейти до п.7.
6) Прийняти *k=4*.
7) Кінець.

Таким чином, при посиланнях вперед безумовно використовуються 4-х байтні команди, незалежно від відстані до мітки.

На другому перегляді всі мітки визначені, тому для визначення посилання вперед використовується порівняння зміщення цільової мітки (*Зм*) плюс значення абсолютного виразу (*Абс*), якщо він є, та значення *$*. Якщо (*Зм+Абс*) > (*$=4*), то безумовно генерується 4-х байтна команда,

інакше – двохбайтна. Для генерації 4-х байтного зміщення команди передачі управління використовується значення виразу *3м-$-4*, а двохбайтного – значення виразу *+3м-$-2*.

5. ЕТАПИ ВИКОНАННЯ ТА ПОРЯДОК ЗАХИСТУ КУРСОВОЇ РОБОТИ

5.1. Графік виконання курсової роботи

Для ефективного планування часу виконання курсової роботи студенту надається календарний план-графік, який містить основні етапи КР та терміни їх виконання (табл. 5.1).

Таблиця 5.1

Календарний план-графік виконання КР

№ пп	Етапи виконання	Термін виконання (тижні семестру)
1.	Отримання студентом індивідуального завдання на виконання курсової роботи	1
2.	Створення на базі варіанту завдання тестових програм мовою Асемблера та узгодження їх з викладачем	2
3.	Розробка лексичного аналізатора	3
4.	Розробка програми 1-го перегляду	7
5.	Розробка програми 2-го перегляду	9
6.	Оформлення результатів курсової роботи	10
7.	Захист курсової роботи	11

Результати виконання кожного з етапів мають бути представлені керівнику у встановлені планом-графіком терміни для поточного контролю роботи студента. Керівник здійснює контроль за ходом роботи, надає студенту необхідну консультативну допомогу при вивченні теоретичного матеріалу та розробленні програмного забезпечення, дає висновок про допуск роботи до захисту.

5.2. Вимоги до оформлення результатів курсової роботи

За результатами виконання курсової роботи і для допуску до захисту КР студент повинен підготувати:

1. Роздруковану та підписану автором **Пояснювальну записку.**
2. Текстовий файл з пояснювальною запискою.
3. Файли початкових модулів компілятора.
4. Виконавчий файл компілятора.
5. Файли тестових програм.

Пояснювальна записка повинна містити:

1. Титульну сторінку з загальноприйнятим вмістом (зразок у Додатку А).
2. Технічне завдання загального змісту, а також з вказівкою номера та змісту індивідуального варіанта.
3. Опис загальної структури розробленої програми, окремих модулів і підпрограм та їх взаємодії.
4. Додаток 1 з текстами початкових модулів компілятора.
5. Додаток 2 з двома тестовими прикладами: перший – без помилок (зразок у Додатку В), другий – з типовими помилками.

5.3. Процедура захисту курсової роботи

До захисту курсової роботи допускаються студенти, які виконали її в повному обсязі та у встановлені терміни і представили відповідні матеріали (див. п. 5.2) керівнику.

У разі недопуску курсової роботи до захисту студент повинен врахувати зауваження керівника курсової роботи та доопрацювати її з повторною подачею керівникові у встановлений термін.

Захист курсової роботи проходить за встановленим графіком захисту.

За тиждень до захисту студент повинен представити керівнику:

- оформлену пояснювальну записку;
- розроблене програмне забезпечення та вихідний код.

Під час захисту студент має продемонструвати працездатність та коректність роботи створеного ним програмного продукту на комп'ютері, використовуючи підготовлені ним тестові файли програм мовою Асемблера (без помилок та з помилками), що засвідчить факт виконання студентом поставлених завдань.

Студенту може бути запропоновано внести незначні зміни у завдання та модифікувати відповідно до них програму. При виявленні під час демонстрації незначних помилок можливе доопрацювання програми студентом на місці.

Після демонстрації роботи студенту можуть бути поставлені запитання з тематики, вмісту та результатів його роботи, на які він повинен дати чітку й обґрунтовану відповідь.

При визначенні оцінки за курсову роботу враховується поточна робота студента над завданням на КР, результати роботи та її захисту, а саме:

- своєчасність отримання завдання на курсову роботу;
- якість виконання окремих етапів КР, дотримання студентом затвердженого календарного плану виконання курсової роботи;
- якість та своєчасність виконання курсової роботи в цілому;
- відповідність оформлення пояснювальної записки нормативним вимогам та технічному завданню;
- наявність програми та засобів приймальних випробовувань для доведення відповідності програми технічним вимогам (послідовність випробовувань, тестові та еталонні файли, засоби порівняння і т.п.);
- відповідність програми технічним вимогам;
- самостійність розробки;
- правильність і аргументованість відповідей на запитання.

За дострокове та якісне виконання окремих етапів виконання курсової роботи та роботи в цілому студенту можуть бути нараховані заохочувальні бали.

СПИСОК РЕКОМЕНДОВАНОЇ ЛІТЕРАТУРИ

1. Зубков С.В. Assembler. Язык неограниченных возможностей. Приложение 2. Команды Intel 80x86. [Текст] / С.В. Зубков – М., 2001.
2. Юров В. Assembler. Учебный курс. Урок 5. Синтаксис ассемблера. [Текст] / В. Юров – СПб. : Питер, 2001.
3. Квиттнер П. Задачи Программы Вычисления Результаты. Гл 5. Языки ассемблеров и ассемблеры. [Текст] / П. Квиттнер – М. Мир, 1980.
4. Лебедев В.Н. Введение в системы программирования. (в части организации вычисления выражений). [Текст] / В.Н. Лебедев – М. Статистика, 1975.

ДОДАТКИ

Додаток А. Зразок оформлення титульного аркуша курсової роботи

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»
Кафедра системного програмування і спеціалізованих комп'ютерних систем

КУРСОВА РОБОТА

з дисципліни «Системне програмування»
на тему: Розробка компілятора програм мовою Асемблера

Студента (ки) ____ курсу _____ групи
напряму підготовки
6.050102 «Комп'ютерна інженерія»
_____
(прізвище та ініціали)
Керівник_____
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Національна оцінка _____
Кількість балів: _____Оцінка: ECTS _____

Члени комісії _____ _____
(підпис) (вчене звання, науковий ступінь, прізвище та ініціали)
_____ _____
(підпис) (вчене звання, науковий ступінь, прізвище та ініціали)
_____ _____
(підпис) (вчене звання, науковий ступінь, прізвище та ініціали

Київ- 20 __рік

Додаток Б. Основні регістри процесора

8-розрядні регістри

Ім'я регістра	Номер[1]	Характеристика
AL	000	**Регістр даних**. Молодша частина AX При використанні цього регістра команди обробки безпосередніх даних та команди пересилки більш компактні та їх виконання потребує менше часу.
CL	001	**Регістр даних**. Молодша частина CX Додатково використовується у командах зсуву для вказання кількості зсувів.
DL	010	**Регістр даних**.. Молодша частина DX
BL	011	**Регістр даних**.. Старша частина BX
AH	100	**Регістр даних**.. Старша частина AX. Даткові використовується як розширення регістра AL у командах множення та ділення 8-розрядних даних
CH	101	**Регістр даних**. Старша частина CX
DH	110	**Регістр даних**. Старша частина DX
BH	111	**Регістр даних**. Старша частина BX

[1] Вказується двійковий код регістра, що використовується в кодах команд

16-розрядні регістри

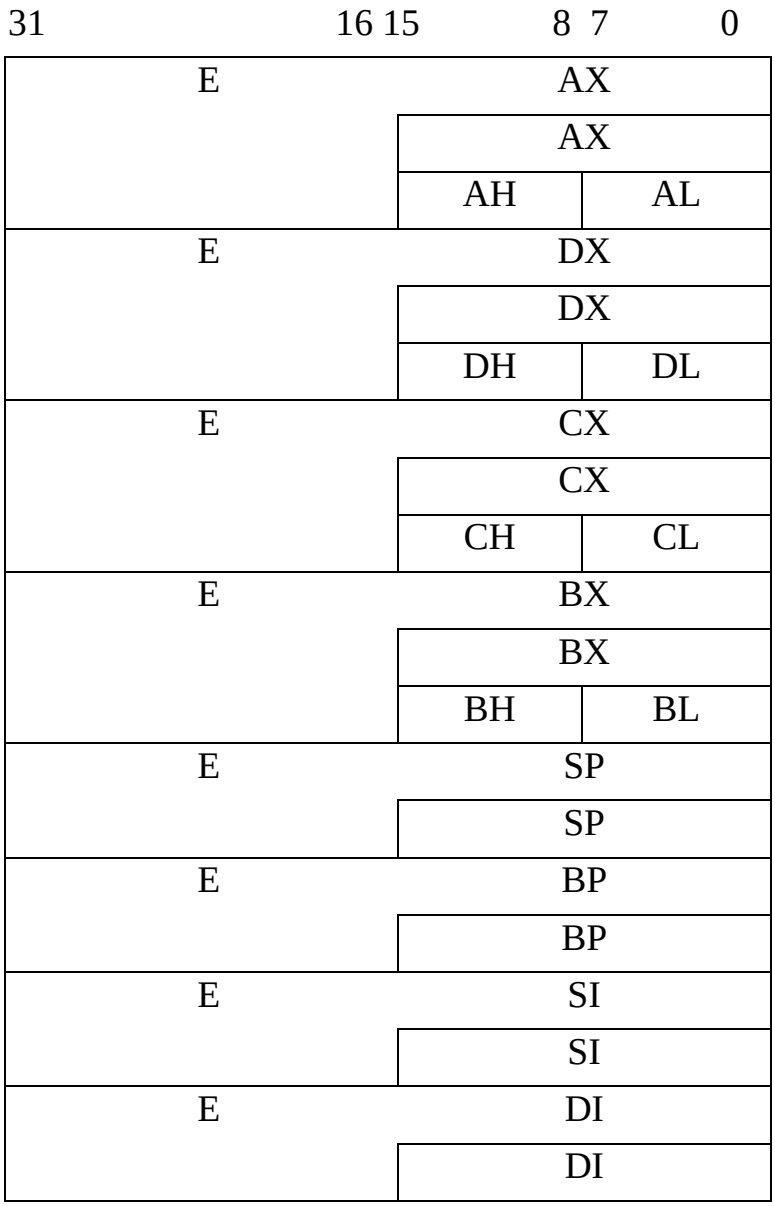
Ім'я регістра	Номер	Характеристика	Сегментний регістр[2]
AX	000	**Регістр даних**. Молодша частина EAX. При використанні цього регістру команди обробки безпосередніх даних та команди пересилки більш компактні та їх виконання потребує менше часу.	-
CX	001	**Регістр даних**. Молодша частина ECX. Додатково використовується як лічильник циклів у рядкових (ланцюгових) командах і командах організації циклів.	-
DX	010	**Регістр даних**. Молодша частина EDX. Додатково використовується як розширення регістру AX у командах множення і ділення 16-розрядних даних Використовується для адресації портів введення-виведення у командах введення та виведення.	-
BX	011	**Регістр загального призначення**. Молодша частина EBX. Типове використання – для адресації складних структур даних.	DS
SP	100	**Вказівник стеку**. Молодша частина ESP. Може використовуватись як **регістр даних.**	SS
BP	101	**Регістр загального призначення**. Молодша частина EBP. Типове використання – для адресації складних структур даних у стеку.	SS
SI	110	**Регістр загального призначення**. Молодша частина ESI. **Особливе** використання – адресація елементів джерела у рядкових (ланцюгових) командах.. Типове використання – адресація елементів масивів.	DS
DI	111	**Регістр загального призначення**. Молодша частина EDI. **Особливе** використання – для адресації елементів рядка приймача у рядкових (ланцюгових) командах. Типове використання – адресація елементів масивів.	DS (ES -у ланцюгових командах)

[2]Вказується сегментний регістр, який використовується за замовчуванням за умови, що відповідний регістр загального призначення використовується як адресний.

32-розрядні регістри

Ім'я регістра	Номер	Характеристика	Сегментний регістр
EAX	000	**Регістр загального призначення.** При використанні цього регістра команди обробки безпосередніх даних і команди пересилки більш компактні та їх виконання потребує менше часу.	DS
ECX	001	**Регістр загального призначення**. Додатково використовується як лічильник циклів у рядкових командах і командах організації циклів	DS
EDX	010	**Регістр загального призначення**. Додатково використовується як розширення регістру EAX у командах множення і ділення 32-розрядних даних.	DS
EBX	011	**Регістр загального призначення.** Типове використання – для адресації складних структур даних.	DS
ESP	100	**Вказівник стеку**. Може використовуватися як **регістр даних**.	SS
EBP	101	**Регістр загального призначення**. Типове використання – для адресації складних структур даних у стеку.	SS
ESI	110	**Регістр загального призначення**. Типове використання – адресація елементів масивів.	DS
EDI	111	**Регістр загального призначення.** Типове використання – адресація елементів масивів	DS (ES – у ланцюгових командах)

Регістрова модель



Сегментні регістри

Складаються з явної 16-розрядної частини та 64-розрядної тіньової. В явну записуються старші 16 біт фізичної адреси сегмента (реальний режим), або селектор дескриптора сегмента (захищений режим). У захищеному режимі, одночасно з завантаженням селектора дескриптора в явну частину, сам дескриптор із пам’яті завантажується у тіньову частину.

Ім'я регістра	Номер	Характеристика	Префікс
ES	000	Містить старші 16 біт фізичної адреси допоміжного сегмента даних (реальний режим) або селектор дескриптора допоміжного сегмента даних (захищений режим). **За замовчуванням використовується для адресації операнда приймача для рядкових (ланцюгових) команд** без можливості заміни. При наявності перед командою префікса 26h (ES:*операнд_пам'яті* мовою Асемблера) використовується замість сегментних регістрів DS та SS, заданих за замовчуванням.	26h (ES:)
CS	001	Містить старші 16 біт фізичної адреси сегмента кодів (реальний режим) або селектор дескриптора сегмента кодів (захищений режим). За замовчуванням використовується для адресації команд без можливості заміни. При наявності перед командою префікса 2Eh (CS:*операнд_пам'яті* мовою Асемблера) використовується замість сегментних регістрів DS та SS, заданих за замовчуванням.	2Eh (CS:)
SS	010	Містить старші 16 біт фізичної адреси сегмента стеку (реальний режим) або селектор дескриптора сегмента стеку (захищений режим). **За замовчуванням** використовується для адресації верхівки стеку у командах **push, pop, call, ret** без можливості заміни. За замовчуванням, з можливістю заміни застосовується в усіх інших командах звернення до пам'яті, де для формування ефективної адреси використовується вміст регістрів **BP, EBP або ESP**. При наявності перед командою префікса 36h (SS:*операнд_пам'яті* мовою Асемблера) використовується замість сегментного регістра DS, заданого за замовчуванням.	36h (SS:)
DS	011	Містить старші 16 біт фізичної адреси сегмента даних (реальний режим) або селектор дескриптора сегмента даних (захищений режим). За замовчуванням, з можливістю заміни, використовується для адресації даних у всіх випадках, коли за замовчуванням не використовуються інші сегментні регістри. При наявності перед командою префікса 3Eh (DS:*операнд_пам'яті* мовою Асемблера) використовується замість сегментного регістра SS, заданого по замовчуванню.	3Eh (DS:)
FS	100	Містить старші 16 біт фізичної адреси сегменту даних (реальний режим), або селектор дескриптору сегменту даних (захищений режим). **За замовчуванням не використовується.** При наявності перед командою префікса 64h (FS:*операнд_пам'яті* мовою Асемблера) використовується замість сегментних регістрів DS та SS, заданих за замовчуванням.	64h (FS:)
GS	101	Містить старші 16 біт фізичної адреси сегмента даних (реальний режим) або селектор дескриптора сегмента даних (захищений режим). **За замовчуванням не використовується.** При наявності перед командою префікса 65h (GS:*операнд_пам'яті* мовою Асемблера) використовується замість сегментних регістрів DS та SS, заданих за замовчуванням.	65h (GS:)

Регістр ознак (Flags)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
			of	df		if	tf	sf	zf	0	af	0	pf	1	cf

cf – ознака перенесення;

pf - ознака парності кількості 1;

af - ознака допоміжного перенесення;

zf – ознака *0* (*0* – число **не дорівнює** *0*, *1* – число **дорівнює** *0*);

sf – ознака знаку числа (*0* – додатне, *1* – від'ємне);

df – ознака напрямку;

of – ознака переповнення.

Додаток В. Приклад тестового файла

```
Data1       Segment
Vb          db          10011b
String      db          ‘Рядок – new’
Vw          dw          4567d
Vd          dd          0d7856fdah
Data1       ends
Data2       Segment
Doublesg    dw          678
QWERTY      dd          67ff89h
Zxcv        db          89h
Data2       Ends
Assume      cs:Code,Ds:Data1,Gs:Data2
Code        Segment
label1:
            Cli
            Inc         cl
            Jb          Label2
            Inc         Bx
            Dec         Vw[si]
            Dec         gs:zxCV[bp]
            Add         Eax, Esi
            Cmp         Ax, Doublesg[edi]
            Cmp         ebx, qwerty[ebx]
            Xor         vb[edx], al
            Mov         dx, 5634h
            Or          Vd[esp], 0101b
            Jb          label1
Label2:
Code        ends
```

ДЛЯ НОТАТОК