МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
КРЕМЕНЧУЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ МИХАЙЛА ОСТРОГРАДСЬКОГО

Ю. В. ЗІЛІНСЬКИЙ, А. Л. ПЕРЕКРЕСТ, А. Л. ЮДІНА

СИСТЕМНЕ ПРОГРАМУВАННЯ. ПРОГРАМУВАННЯ НА АСЕМБЛЕРІ.

НАВЧАЛЬНИЙ ПОСІБНИК

КРЕМЕНЧУК 2023

УДК 681.127

Рекомендовано до друку вченою радою
Кременчуцького національного університету імені Михайла Остроградського
(протокол № 9 від 29 червня 2023 р.)

Рецензенти:
Вовна О. В. – доктор технічних наук, професор
Лактіонов І. С. – доктор технічних наук, професор
Шевченко І. В. – доктор технічних наук, професор

Зілінський Ю. В., Перекрест А. Л., Юдіна А. Л. Системне програмування. Програмування на ассеблері: навч. Посібник. Кременчук: Кременчуцький національний університет імені Михайла Остроградського, 2023. 258 с.

ISBN 978-617-7304-14-1

У навчальному посібнику розглянуто програмну модель мікропроцесорів з архітектурою Intel64/AMD64, методи обробки структурованих типів даних, процедурне та модульне програмування, макрозасоби Microsoft Macro Assembler, програмування математичного співпроцесора і розширення MMX, зосереджено увагу на використанні асемблерних модулів для розробки програмних додатків засобами Microsoft Visual Studio C/C++.

Навчальний посібник містить приклади програмування, а також контрольні питання та тести для самостійної роботи. Навчальний посібник рекомендований для студентів усіх форм навчання спеціальності 123 – «Комп'ютерна інженерія».

УДК 681.127

@ Ю. В. Зілінський, А. Л. Перекрест, А. Л. Юдіна
© Кременчуцький національний університет
імені Михайла Остроградського, 2023

ЗМІСТ

ВСТУП

Системне програмування передбачає керування апаратними складовими комп'ютерних систем і організацію функціонування їхніх програмних засобів.

Завданнями системного програмування є розробка Base Input Output System (BIOS) та Unified Extensible Firmware Interface (UEFI), операційних систем (ОС), драйверів, засобів віртуалізації апаратних і програмних середовищ, засобів розробки програм (трансляторів, лінкерів, компіляторів, інтерпретаторів, налагоджувачів, дизасемблерів), програм обслуговування і діагностики апаратної частини, систем захисту від несанкціонованого доступу та шкідливих програм.

Кожне із цих завдань є окремим напрямом розробки системного програмного забезпечення і може бути предметом вивчення окремого багатогодинного навчального курсу. Деякі з цих напрямів є суміжними або взаємозалежними.

Традиційно мовами програмування у системному програмуванні вважаються С і асемблер.

Навчальний посібник призначений сприяти вивченню мови асемблера мікропроцесорів Intel64/AMD64.

Навчальна дисципліна «Системне програмування» належить до дисциплін професійної підготовки бакалаврів зі спеціальності 123 – «Комп'ютерна інженерія». Навчальний посібник може бути використаний студентами всіх форм навчання в процесі самостійної підготовки до занять з навчальної дисципліни «Системне програмування» і під час вивчення інших навчальних дисциплін, які передбачають розробку низькорівневого програмного забезпечення.

1 ЗАСОБИ РОЗРОБКИ

1.1 Склад засобів розробки

Системні програми тісно пов'язані з конкретною апаратною платформою комп'ютера і деяким чином із середовищем (підсистемою виконання) певної операційної системи.

У навчальному посібнику розглянуто питання розробки 64-розрядних програмних модулів для 64-розрядних версій ОС Microsoft Windows NT на платформі Intel64/AMD64 (x86_64) та організації їх взаємодії з програмними модулями розробленими з використанням Microsoft Visual C/C++. Із урахуванням цього основою засобів розробки вибрано асемблер Microsoft Macro Assembler від розробника самої ОС Windows NT компанії Microsoft.

Основу використовуваних засобів розробки складають:

1. Microsoft Macro Assembler зі складу Microsoft Visual Studio Community (https://visualstudio.microsoft.com/ru/free-developer-offers/) і MASM64 SDK (http://masm32.com/board/index.php?topic=10052.0) by Steve Hutchesson.
2. Налагоджувач x64dbg (https://x64dbg.com/) та/або засоби налагодження зі складу SDK/WDK (https://learn.microsoft.com/en-us/windows-hardware/drivers/download-the-wdk).
3. Інтегроване середовище розробки програм RadAsm IDE (https://web.archive.org/web/20130525185745/http://assembly.com.br/).

1.2 Склад асемблера Microsoft Macro Assembler (MASM)

Асемблери являють собою набір (пакет) програм і допоміжних файлів, які дозволяють отримувати різноманітні програмні модулі для *певної процесорної архітектури* та середовища (підсистеми виконання) певної операційної системи.

Засоби асемблера також дозволяють розробляти програми, які працюють без операційних систем, наприклад BIOS або UEFI, які починають працювати ще до, або на етапі завантаження операційної системи. Власне і сама операційна система також може бути розроблена за допомогою асемблера. *Основу будь-якого асемблера складають транслятор і компанувальник.*

Транслятор (ml64.exe) – програма для перетворення вихідного тексту програми на об'єктний файл (код), що містить машинні коди команд мікропроцесора. Під терміном «асемблер» здебільшого розуміють саме транслятор, а систему позначень, яка використовується для запису вихідного тексту (коду) програми для транслятора, називають *мовою асемблера*.

Компанувальник (редактор зв'язків, лінкер) link.exe – програма для перетворення об'єктного файлу (коду) на програмний модуль, який може виконуватися в певному середовищі або певною підсистемою виконання операційної системи.

Ці та інші програмні засоби традиційно являють собою консольні додатки з інтерфейсом командного рядка, що не завжди є зручним. Роботу з консольними додатками в зручному графічному інтерфейсі забезпечують ріноманітні інтегровані середовища розробки (англ. Integrated Development Environment, IDE). Авторами обрано RadASM IDE by Ketil Olsen (https://web.archive.org/web/20130525185745/http://assembly.com.br/).

1.3 Склад і взаємозв'язок файлів проєкту

Для розробки програмного забезпечення використовують термін проєкт, під яким зокрема розуміють набір пов'язаних поміж собою текстових і бінарних файлів. Мінімальний проєкт програми на асемблері може складатися з єдиного файлу з вихідним текстом програми. Традиційно цей файл має розширення «asm».

До складу проєкту також можуть входити файли з розширеннями «inc» і «mac», що підключаються у файлах проєкту за допомогою спеціальних директив.

Розширення цих файлів, власне не має особливого значення, але традиційно mac-файли містять так звані макровизначення, а inc-файли – це так звані заголовні файли, що містять опис типів, прототипів, структур, констант тощо.

Під час розробки програм, особливо з використанням елементів графічного інтерфейсу ОС Windows, до складу проєкту, зазвичай, також входять так звані файли ресурсів. Файли ресурсів являють собою текстові файли з розширенням «rc», що містять спеціальні директиви опису використовуваних програмою елементів графічного інтерфейсу, зрозумілі компілятора ресурсів rc.exe.

Мову опису ресурсів знати корисно, однак на практиці для розробки файлів ресурсів широко використовують програмні *засоби підтримки візуального програмування – редактори ресурсів*.

Файли ресурсів можуть посилатися на бінарні (двійкові) файли іконок, курсорів, шрифтів, та ін. За допомогою компілятора ресурсів текстові файли ресурсів перетворюються на бінарні файли з розширенням «res», які в подальшому використовується компанувальником для складання проєкту і потрапляють до складу програмного модуля.

У деяких випадках (зазвичай під час створення програмних модулів відмінних від звичайних «exe»-файлів) для складання проєкту компанувальник потребує певної додаткової інформації, яка міститься у так званому файлі визначення програмного модуля. Цей текстовий файл з розширенням «def» містить зрозумілі компанувальнику спеціальні директиви, які керують його роботою. Ім’я def-файлу передається компанувальнику за допомогою спеціальних параметрів командного рядка.

1.4 Етапи розробки проєкту

Розробка проекту починається зі створення всіх вихідних файлів проєкту, які, зазвичай, розміщують в одному каталозі. Створення і редагування текстових файлів можна виконувати у будь-якому текстовому редакторі, але з певними застереженнями щодо використання текстових даних у різних кодуваннях.

Рисунок 1.1 демонструє у вигляді граф-схеми етапи розробки проєкту, які позначено цифрами над дугами граф-схеми.

Після створення проєкту та редагування всіх потрібних вихідних файлів зазвичай спочатку виконують компіляцію файлів ресурсів за допомогою компілятора ресурсів rc.exe. Компіляцію файлів ресурсів може бути виконано і після трансляції модулів з вихідним текстом програми, але на етапі налагодження програми, файли ресурсів доводиться, зазвичай, редагувати щонайменше.

Отримавши відразу бінарний файл з ресурсами, його потім багаторазово використовують під час повторного компонування програми у разі змін у тексті інших вихідних файлів.

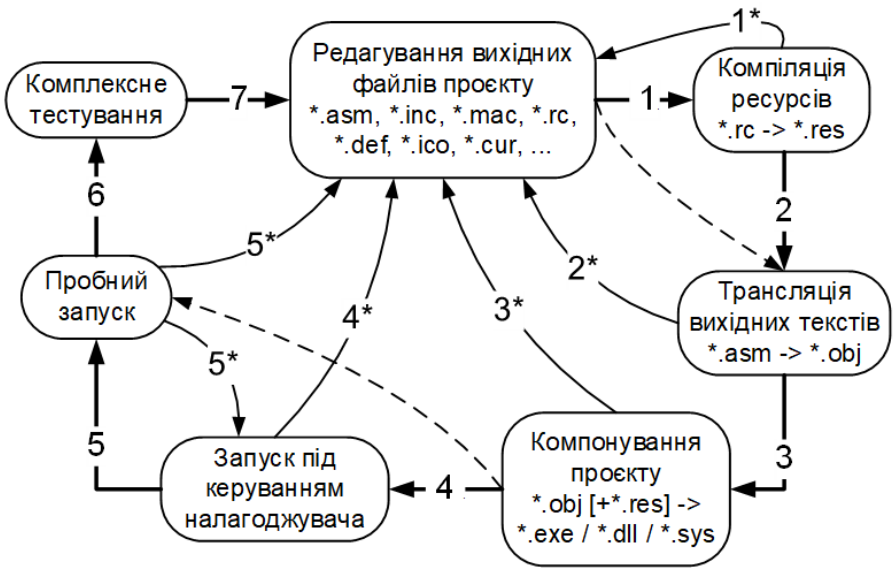


Рисунок 1.1 – Етапи розробки проєкту

Файли ресурсів можуть містити помилки, для виправлення яких, можливо, знадобиться багаторазове повернення до редагування цих файлів з повторною компіляцією.

Після компіляції ресурсних файлів розпочинають до трансляцію вихідних текстів модулів програми. Для програм, які не використовують rc-файли, відразу виконується трансляція вихідних текстів модулів та/або програми.

Якщо вихідні тексти файлів проєкту не мають помилок, то трансляція закінчується створенням об'єктних модулів – файлів з розширенням «obj». Проєкт може складатися з декількох asm-файлів і, відповідно, з декількох отриманих після їх трансляції об'єктних модулів. Асемблери припускають складання програмного файлу з різних отриманих раніше об'єктних файлів, що можуть використовуватися у різних проєктах уже не на рівні вихідного тексту, а у вигляді готових obj-бібліотек.

Якщо всі бінарні та об'єктні файли отримано, виконується складання проєкту із цих файлів за допомогою компанувальника. Отримання obj-файлів є необхідною умовою, але в деяких випадках недостатньою для того, щоб отримати програмний модуль. У вихідному тексті можуть бути директиви, які не важливі для транслятора, але мають значення для компанувальника, і у разі його повідомлення про неможливість з певних причин створити програмний модуль, доводиться повертатися назад до редагування вихідного тексту та повторної трансляції.

Важко сподіватися на те, що більш-менш серйозна програма відразу запрацює правильно. Отже, етап налагодження завжди неминучий і, до того ж, найдовший. Для підвищення ефективності процесу налагодження програм транслятор і компанувальник мають спеціальні засоби. Організація їх взаємодії з налагоджувачем виконується за допомогою відповідних параметрів командного рядка.

Параметр командного рядка «/DEBUG» примушує компонувальник формувати інформацію для налагоджувача, створюючи спеціальні файли з розширенням «pdb». Коли процес налагодження (нарешті!) закінчено і потреби в інформації налагоджування не існує, то фінальний варіант програмного файлу потрібно отримувати у «чистому» вигляді, змінивши параметр командного рядка

компанувальника «/DEBUG» на «/RELEASE». Слід зауважити, що наявність налагоджувальної інформації спрощує процес його дизасемблювання сторонніми особами.

У результаті роботи транслятора та компанувальника бінарна інформація для налагоджувача і текстова для розробника створюється у вигляді окремих файлів (*.pdb, *.lst, *.map). Створення цих файлів потрібно замовляти відповідними параметрами командного рядка під час виклику транслятора і компанувальника.

Після отримання програмного модуля виконується запуск програми під керуванням налагоджувача. Можна, але не варто безпосередньо розпочинати пробний запуск програми. Пробний запуск програми може допомогти виявити і певним чином визначити місця виникнення помилок, але для їх локалізації все одно доведеться користуватися послугами налагоджувача, і, до того ж, зовнішні прояви помилок можуть з'являтися далеко від місця їх фактичного виникнення.

Виправлення помилок супроводжується редагуванням вихідних текстових файлів, і, звичайно, повторною трансляцією і компануванням. Після виправлення всіх явних помилок знову виконується пробний запуск програми і починається процес комплексного тестування.

1.5 Структура вихідного тексту програми для асемблера MASM

Вихідний текст програми на асемблері складається з *директив, команд, макрокоманд і коментарів*. У записах директив, команд і макрокоманд можуть використовуватися *оператори асемблера*. Традиційно більшість асемблерів, серед яких і MASM, підтримують запис *єдиної* директиви, команди або макрокоманди в одному рядку вихідного тексту програми.

Ознакою коментаря є символ «;» (крапка з комою), виявивши який у будь-якому місці вихідного тексту, транслятор припиняє його обробку *до кінця рядка* за цим символом. Така поведінка транслятора обумовлює те, що коментарі

розміщують або з початку нового рядка, або справа від директив, команд і макрокоманд в одному з ними рядку.

Для багаторядкових коментарів зручно використовувати директиву

comment обмежувач

……..

……..

обмежувач

Як обмежувач можна використовувати будь-яке слово з припустимих символів алфавіту асемблера, за умови, що це слово не міститься в тексті самого коментаря, а однаковий обмежувач можна використовувати багаторазово *в різних* директивах comment.

Гарна звичка коментувати вихідний текст своїх програм повністю компенсує витрачений для цього час потім, на етапі їх супроводження, допоможе підвищити їх якість і безпечність завдяки подальшій ревізії коду (англ. Code review або Code inspection).

1.6 Вихідний текст 64-розрядного додатка ОС Windows NT

Вихідний текст 64-розрядного додатка ОС Windows NT для Microsoft Macro Assembler має такий вигляд (у квадратні дужки [] взято необов'язкові елементи):

[option опція]

……

[include [<шлях>\]ім'я.inc]

……

[includelib [<шлях>\]ім'я.lib]

……

[<макровизначення>]

[.const]

[<директиви визначення та ініціалізації даних (констант)>]

```
 [.data]
[<директиви визначення та ініціалізації даних>]
[.data?]
[<директиви визначення даних і резервування пам’яті>]
.code
[<визначення підпрограм>]
main proc
        [<код>]
        ret
main endp
[<визначення підпрограм>]
end
```

Директиви option дозволяють зазначити додаткові параметри трансляції вихідного тексту програми.

Директиви include [<шлях>\]ім’я.inc використовуються для підключення до вихідного тексту програми, загалолм будь-яких текстових файлів, синтаксис яких зрозумілий для транслятора. Зазвичай це файли з розширеннями «mac», що містять так звані макровизначення, і файли з розширеннями «inc», що містять опис типів, прототипів, структур, констант тощо. Окрім власних файлів з макровизначеннями і заголовних файлів, проєкт може використовувати також сторонні заголовні файли та файли з макровизначеннями. Зокрема використовуватимемо файли зі складу MASM64 SDK.

Виявивши в програмі директиву include, транслятор відкриває заданий за допомогою неї файл і починає його обробку від початку до кінця так, немов би це продовження вихідного тексту програми, а лише потім розпочинає обробку наступного рядка вихідного тексту.

У технологічному лістингу програми, який є робочим варіантом вихідного тексту програми для транслятора, замість кожної директиви include вставляється текст того файлу, на який йде посилання. Якщо водночас ці файли містять власні

директиви include, то їх обробка виконується так само. Лише отримавши такий повний текст, складений з різних файлів, асемблер розпочинає його трансляцію.

Мати повний лістинг програми корисно. Його створення потрібно замовляти параметром командного рядка транслятора /Fl і, окрім того, асемблер має спеціальні засоби керування лістингом програми. Аналіз лістингу дозволяє детальніше розібратися у структурі програми та виявити помилки, спричинені макровизначеннями. На етапі обробки рядків з include асемблер контролює вміст inc-файлів і виводить попереджувальні повідомлення, або навіть може зовсім припинити обробку у разі виявлення помилок.

Аналіз причини помилки можна здійснити на підставі тексту файлу лістингу. Спочатку треба перевірити правильність задання місцеположення inc-файлів на диску. Параметр <шлях> директиви include хоча і позначений квадратними дужками як необов'язковий, але якщо його не задати у вихідному тексті програми, це все одно доведеться зробити, але вже у командному рядку транслятора за допомогою параметра /I:<шлях>. Перевірки шляхів доступу також потрібно виконати у «продовженнях» inc-файлів, що містять власні директиви include.

Директиви includelib [<шлях>\]ім'я.lib задають перелік усіх файлів бібліотек імпорту і статичних бібліотек, які знадобляться компонувальнику для збирання програмного модуля. Методи зв'язування із зовнішніми бібліотеками буде детально розглядати пізніше у розділі «Модульне програмування», а наразі лише зауважимо, що параметр <шлях> хоча є необов'язковим, але якщо його не зазначити у тексті програми, то це все одно доведеться зробити за допомогою відповідного параметра командного рядка лінкера /LIBPATH:<шлях>.

Директиви .const, .data, .data?, .code – це директиви керування секціями програми. Код програми являє собою зв'язний список машинних команд, тобто команди програми йдуть одна за іншою в строго певному порядку, що задається алгоритмом. Тому цілком природно, що зберігати цей зв'язний список зручніше в компактному вигляді, відокремивши код машинних команд від даних,

розміщуючи при завантаженні в пам’ять програми для її виконання *команди й дані за різними адресами*. Отже, визначимо суть поняття секції.

Секція – це ділянка (діапазон адрес) пам’яті, що підтримується на програмно-апаратному рівні (кодом операційної системи й апаратурою мікропроцесора) і використовується для розміщення в пам’яті *різних за функціональним призначенням* частин програми.

Ділянку пам’яті, у якій розташовуються коди машинних команд, називають секцією коду, а ділянку, у якій розташовуються дані програми, називають секцією даних. Завжди в пам’яті під час роботи програми наявна ще одна секція – стек. Розміщує та підтримує структуру стека операційна система, і розробнику програми немає потреби зосереджуватися на цьому.

Для логічного розподілу тексту програми на окремі фрагменти згідно з їх функціональним призначенням і для позначення початку кожної секції використовуються директиви .const, .data, .data?, .code. Початок нової секції автоматично означає кінець попередньої. Послідовність секцій не має значення, а кількість їх вибирає автор програми за власними потребами – транслятор і компанувальник забезпечують *об’єднання однакових секцій під час збирання програмного модуля*.

Усе, що розташовується після директиви .code і до директиви end, асемблер намагатиметься транслювати у машинний код. Для позначення того місця в коді, із якого починається виконання програми використовується директива proc. Ім’я main у директиві може вибиратися довільно, але має збігатися з іменем у директиві endp. Таким чином пара директив main proc ... main endp використовується для позначення основного коду програми.

Після завантаження програми в пам’ять її виконання почнеться з першою машинною командою після main proc і завершиться машинною командою ret, яка поверне керування операційній системі.

Після директиви .code, але до main proc, а також після main endp, але до директиви end, розміщуються підпрограми, які використовує програма в процесі

своєї роботи. Таке розміщення забезпечує трансляцію вихідного тексту підпрограм у машинний код і не припускає несанкціонованого (якщо програма сама не ініціює виклик підпрограм) виконання коду підпрограм.

Секція .code, загалом може містити не тільки код, а і дані програми, які можна, наприклад, розміщувати в тому ж місці, що і підпрограми. Розміщення даних у секції коду без особливої потреби недоцільне і має обмеження, оскільки *для секції коду в загальному випадку не передбачена можливість запису*.

У секції .data розміщуються дані, які програма використовуватиме в процесі своєї роботи. Ці дані визначаються в секції .data розробником програми і доступні як для читання, так і для запису. Усе, що визначено в секції .data, *розміщується у програмному файлі* і завантажується в пам'ять із завантаженням програми на виконання.

Спроби резервувати місце в секції .data для ще неіснуючих (невідомих) даних, які з'являться лише в процесі роботи програми (є результатом її роботи) призводять до збільшення розміру програми на диску, і щоб цьому запобігти, у структурі програми передбачено наявність спеціальної секції неініціалізованих даних, початок якої позначається за допомогою директиви .data?.

Пам'ять для секції неініціалізованих даних .data? програма отримає лише на етапі її завантаження на виконання, ці дані *не займають місця у програмному файлі*. У іншому секція неініціалізованих даних нічим не відрізняється від секції ініціалізованих даних, для неї також припустимо використання операцій як запису, так і читання.

Секція, що визначається директивою .const, як указує її назва, слугує для розміщення констант, тобто незмінних (сталих) даних, які використовує програма у процесі своєї роботи. Усі дані секції констант *розміщуються у програмному файлі*, і завантажується в пам'ять із завантаженням програми на виконання.

Секція .const може використовуватися для розміщення важливих програмних даних з метою їх захисту від помилкової модифікації. Для секції

.const так само як і для секції .code, *не передбачена можливість запису*, що і має забезпечувати незмінність визначених у секції .const даних.

Альтернативним, щодо використання секції .const, методом розміщення сталих даних можна вважати їх розміщення у секції коду .code, що також певним чином гарантує їх «недоторканість».

1.7 Трансляція і компонування проєкту

Якщо позбутися всіх необов'язкових елементів, узятих у квадратні дужки у вихідному тексті програми, наведеної в попередньому розділі, то отримаємо мінімальний вихідний код найпростішого, але повністю працездатного, додатка для 64-розрядних ОС Windows NT.

Для отримання файлу з іменем ProjectName.asm, що містить вихідний код додатка можна скористатися *будь яким текстовим редактором*, наприклад Блокнот (notepad.exe), який є стандартним додатком усіх версій Windows.

Вміст файлу ProjectName.asm 64-розрядного додатка:

```
.code
main proc
        ret
main endp
end
```

Транслятор ml64.exe та компанувальник link.exe являють собою консольні додатки з інтерфейсом командного рядка. Для більш зручного використання транслятора та компанувальника з метою отримання програмного файлу додатка краще додати повний шлях до підкаталогів їх розташування в змінну оточення %PATH%. Після цього потрібно виконати запуск командного інтерпретатора cmd.exe, перейти в каталог з вихідними файлами проєкту і виконати такі дії:

1. Трансляція (ProjectName.asm → ProjectName.obj):

   ML64.EXE /c /Cp /nologo "ProjectName.asm"

2. Компонування (ProjectName.obj → ProjectName.exe):

LINK.EXE /ENTRY:main /MACHINE:X64 /SUBSYSTEM:CONSOLE /DEBUG "ProjectName.obj"

Інтерфейс командного рядка не завжди є зручним, і в подальшому для розробки проєктів краще використовувати інтегроване середовище розробки, яке крім графічного інтерфейсу забезпечує певні зручності, як от: довідкова система, готові шаблони різних типів проектів, можливість інтеграції із зовнішнім налагоджувачем і додатковими утилітами.

2 ПРОГРАМНА МОДЕЛЬ БАЗОВОГО МІКРОПРОЦЕСОРА

Як базову модель розглядаємо мікропроцесори з архітектурою Intel64 (не IA-64!) компанії Intel та сумісні з ними мікропроцесори з архітектурою AMD64 компанії AMD. Інші назви (IA-32-EM64T, x86-64) є синонімами Intel64/AMD64.

Intel64/AMD64 є нащадком попередньої архітектури IA-32 (x86) і зворотно сумісна з нею, тобто мікропроцесори з архітектурою Intel64/AMD64 можуть працювати так само як їх пращури, із архітектурою IA-32, не використовуючи свої нові можливості.

Під програмною моделлю мікропроцесора розуміють його організацію щодо можливостей програмування. Складові програмної моделі мікропроцесорів подано на рисунку 2.1.

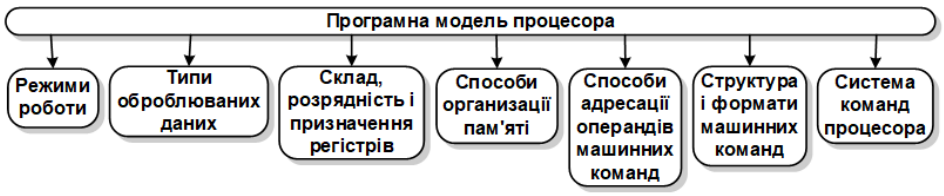


Рисунок 2.1 – Програмна модель мікропроцесора

2.1 Режими роботи

Від режиму роботи процесора залежить інтерпретація усіх інших складових програмної моделі, окрім хіба що типів оброблюваних даних. Рисунок 2.2 відображає режими роботи Intel64/AMD64 у вигляді графа. Стрілки

на графі показують можливості перемикання процесора з одного режиму в інший.

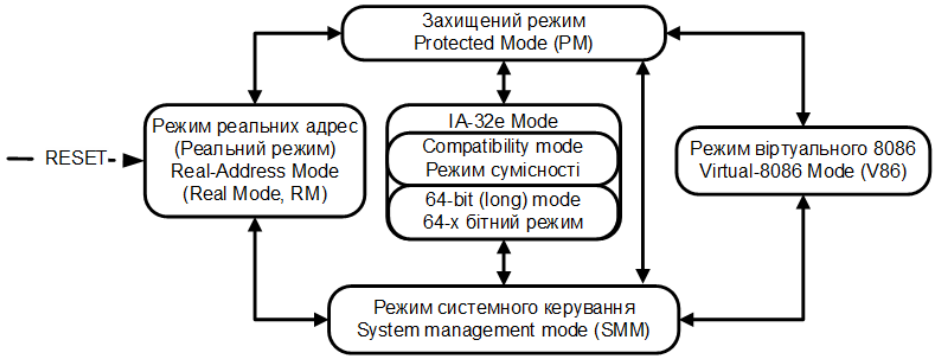


Рисунок 2.2 – Режими роботи Intel64/AMD64

Архітектура Intel64/AMD64 розширила попередню архітектуру IA-32 новим режимом IA-32e. Усі нові можливості Intel64/AMD64 доступні тільки в «довгому» 64-бітному режимі, інакше Intel64/AMD64 ідентичний IA-32. У подальшому все стосуватиметься 64-бітного «довгого» під- режиму IA-32e.

2.2 Типи оброблюваних даних

2.2.1 Апаратно підтримувані типи даних

Апаратно підтримувані типи даних обумовлюються розрядністю апаратних складових мікропроцесора. Апаратно (на фізичному рівні) мікропроцесор підтримує такі типи даних:

1. Байт.
2. Слово = 2 байти = 16 біт.
3. Подвійне слово = 2 слова = 4 байти = 32 біта.
4. Зчетверене слово = 2 подвійні слова = 4 слова = 8 байт = 64 біта.
5. Подвійне зчетверене слово = 2 зчетверених слова = 4 подвійні слова = 8 слів = 16 байт = 128 бітів.

Більші, ніж байт, типи даних прийнято ділити на дрібніші складові, відповідно до схеми на рисунку 2.3.

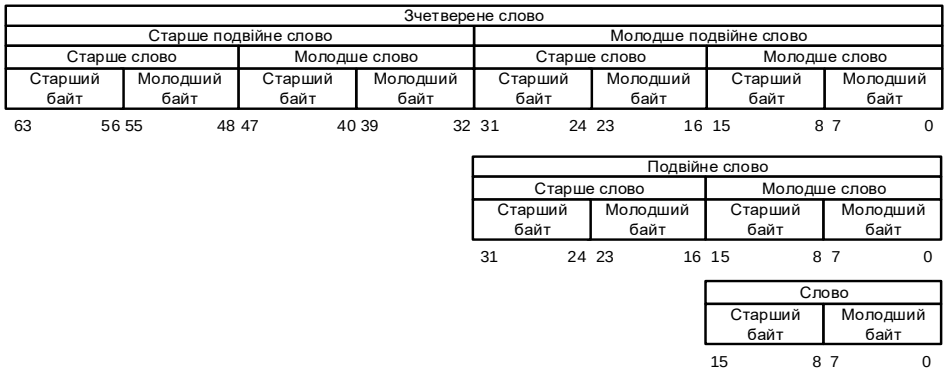


Рисунок 2.3 – Апаратно підтримувані типи даних

У деяких випадках розподілу підлягають і байти. Біти 7 – 4 називають старшим напівбайтом або старшою тетрадою, а біти 3 – 0, відповідно, – молодшим напівбайтом або молодшою тетрадою.

У деяких випадках *для розширення діапазону подання* апаратно підтримуваних мікропроцесором типів даних вони поєднуються в пари <старше слово:молодше слово >, <старше подвійне слово:молодше подвійне слово>, <старше зчетверене слово:молодше зчетверене слово>.

2.2.2 Логічна інтерпретація апаратних типів даних

Мікропроцесор дотримується принципу *байдужності до цільового призначення даних,* і «апаратно підтримувані» означає, що послідовності біт зазначених вище розмірів можуть брати участь як операнди виконуваних мікропроцесором операцій. Про те, що позначають оброблювані послідовності біт, процесор не здогадується, і для нього це байдуже.

Логічна інтерпретація типів даних програми й результатів її роботи важлива для розроблювача/користувача, і він може їх трактувати в такий спосіб:

1. Цілий тип зі знаком – двійкове значення зі знаком, розміром 8, 16, 32 або 64 біта *подане у додатковому коді* (7, 15, 31 або 63-й біт знаковий):
    а) 8-розрядне ціле – від –128 до +127;
    б) 16-розрядне ціле – від –32 768 до +32 767;
    в) 32-розрядне ціле – від –2 147 483 648 до 2 147 483 647;
    г) 64-розрядне ціле – від -2^{63} до $+2^{63}-1$.
2. Цілий тип без знака – двійкове значення без знака, розміром 8, 16, 32 або 64 біта:
    а) 8-розрядне без знакове ціле – від 0 до 255;
    б) 16-розрядне без знакове ціле – від 0 до 65 535;
    в) 32-розрядне без знакове ціле – від 0 до 4 294 967 295;
    г) 64-розрядне без знакове ціле – від 0 до $2^{64}-1$.
3. Символьний тип – двійкове значення розміром 8 або 16 біт, що являє собою код символу в певній кодовій таблиці:
    а) символ у байтовому кодуванні (ASCII, UTF-8, KOI8-U і т.д);
    б) символ у двохбайтовому кодуванні Unicode (UTF-16).
4. Двійково-десятковий тип:
    а) непаковане двійково-десяткове число;
    б) паковане двійково-десяткове число.
5. Покажчик на пам’ять.
6. Типи даних математичного співпроцесора.
7. Типи даних MMX.
8. Типи даних XMM.

2.2.3 Директиви визначення даних і резервування пам’яті

Для визначення значень змінних простих типів у секції ініціалізованих даних .data і секції констант .const використовується конструкція виду:

[ім’я змінної] Директива Значення

Ім’я змінної, *так само, як і будь яке інше символьне ім’я використовуване в програмі на асемблері*, повинно бути довжиною до 247 символів a–z, A–Z, 0–9, «@», «_» «$» «?» і не повинно починатися із символу цифри 0–9.

Як директива використовується одне з позначень, наведених у таблиці 2.1.

Таблиця 2.1 – Директиви визначення даних і резервування пам’яті

Директива*	Розмір, байт	Синонім*	Призначення
DB	1	BYTE	8-розрядне беззнакове ціле від 0 до 255, символ у байтовому кодуванні
		SBYTE	8-розрядне ціле від –128 до +127
DW	2	WORD	16-розрядне беззнакове ціле – від 0 до 65 535, символ у двобайтовому кодуванні
		SWORD	16-розрядне ціле від –32 768 до +32 767
DD	4	DWORD	32-розрядне беззнакове ціле від 0 до 4294967295
		SDWORD	32-розрядне ціле від –2147483648 до 2147483647
		REAL4	коротке 32-бітне дійсне число FPU
DF	6	FWORD	48-розрядний покажчик на пам'ять
DQ	8	QWORD	64-розрядне беззнакове ціле від 0 до 2^{64}–1, 64-розрядний покажчик на пам’ять
		SQWORD	64-розрядне ціле від -2^{63} до $+2^{63}$–1
		REAL8	довге 64-бітне дійсне число FPU
		MMWORD	64-розрядний операнд команд розширення MMX
DT	10	TBYTE	паковане двійково-десяткове число
		REAL10	80-бітне дійсне число в розширеному форматі FPU
	16	XMMWORD	128-розрядний операнд команд розширення SSE
	32	YMMWORD	256-розрядний операнд команд розширення AVX

* – регістронезалежні, тобто припустимо, наприклад, використання DB, db, BYTE, byte.

Значення ініціалізації може задаватися безпосереднім значенням, тобто числом. Асемблер підтримує запис чисел у двійковій, десятковій та шістнадцятковій системах числення. Для явного зазначення системи числення для запису числа використовуються кінцеві символи «b»|«B» для двійкових чисел, «h»|«H» – для шістнадцяткових, та «d»|«D» – для десяткових. *Десяткова система числення використовується за замовчанням*, якщо в тексті програми немає відповідних директив перевизначення. Якщо шістнадцяткове число починається із символу літери, то перед ним обов'язково необхідно задавати цифру 0. Наприклад:

```
a1      db      10
a2      BYTE    10d
a4      DB      0Ah
a5      BYTE    1010b
```

Слід пам'ятати, що під час визначення негативних чисел у десятковій системі числення у пам'яті вони будуть подані в двійковому додатковому коді, а під час визначення негативних чисел у двійковій та шістнадцятковій системі числення слід самостійно виконувати перетворення на додатковий код. Наприклад:

```
a1      DB      -10         ; = 11110110b = 0F6h
a2      db      0F6h        ; -10  як знакове (246 як беззнакове)
a3      BYTE    11110110b   ; -10  як знакове (246 як беззнакове)
```

Значення ініціалізації також може задаватися виразом, побудованим з використанням символьних імен інших об'єктів програми і операторів асемблера. Однак така можливість поки що не розглядається.

Для визначення масивів з елементів простих типів у секції ініціалізованих даних .data і секції констант .const використовується конструкція виду

```
[ім'я масиву]   Директива   Значення1, Значення2,…, ЗначенняN
```

Наприклад,

```
array    db    1, 2, -3, 0100b, -5, 6d, 7, 1000b, -9, 0Ah
```

Визначення рядків символів у секції ініціалізованих даних .data і секції констант .const може виконуватися у вигляді масивів байтів/слів або з використанням операторів «' '» чи «” ”». Слід пам’ятати, що асемблер інтерпретує символи як числа, що дорівнюють відповідним їм кодам. Наприклад:

```
str_1        db "ABCD"
str_2        db 'ABCD'
str_3        db 'A','B','C','D'
str_4        db "A","B","C","D"
str_5        db 41h, 42h, 43h, 44h
str_6        db 'A', "B", 43h,01000100b
```

Для визначення в секції ініціалізованих даних .data і секції констант .const масивів з однаковими значеннями елементів або рядків з одного і того самого символу використовується конструкція виду

[ім’я] Директива Кількість DUP (Значення)

Наприклад,

```
Handles    QWORD    64    dup (-1)
Line       db       80    dup ("*")
```

Резервування пам’яті під подальше розміщення даних простих типів, масивів з елементів простих типів і рядків символів у секції неініціалізованих даних .data? відрізняється від їх визначення в секції ініціалізованих даних .data і секції констант .const лише тим, що *значення ініціалізації завжди позначається символом* «?». Наприклад:

```
a1         db       ?
array1     db       ?, ?, ?, ?, ?, ?, ?, ?, ?, ?
array2     db       10    dup (?)
str_1      db       4     dup (?)
Handles    QWORD    64    dup (?)
```

За допомогою директиви TYPEDEF і директив визначення даних і резервування пам’яті можна визначати власні назви апаратно підтримуваних типів даних. За типами, визначеними за допомогою директиви TYPEDEF, можна визначати нові типи. Наприклад:

bool	typedef	SBYTE
char	typedef	SBYTE
unsigned_char	typedef	BYTE
signed_char	typedef	SBYTE
unsigned_short	typedef	WORD
unsigned_int	typedef	DWORD
long	typedef	SDWORD
unsigned_long	typedef	DWORD
float	typedef	REAL4
double	typedef	REAL8

Визначення типів повинно виконуватися до їх першого використання в програмі. Зазвичай для цього використовуються заголовні файли.

2.2.4 Особливості розміщеннями даних у пам’яті

Внутрішня архітектура мікропроцесорів Intel64/AMD64 обумовлює певні особливості розміщеннями ними даних у пам’яті, про які потрібно пам’ятати під часрозробки та налагодження програм.

Існує два порядки зберігання байтів машинних слів для запису їх у пам’ять. Порядок від молодшого до старшого – little-endian (або «гострокінцевий»), за якого запис у пам’ять починається з молодшого байта й закінчується старшим, і порядок від старшого до молодшого – big-endian (або «тупокінцевий») за якого запис в пам’ять починається зі старшого байта й закінчується молодшим.

Мікропроцесори Intel64/AMD64 використовують little-endian спосіб розташування, при якому машинні слова розміром, кратним 2, зберігаються в пам’яті в перевернеmому вигляді, а адресою машинного слова вважається адреса

його наймолодшого байта. Надалі назвемо це правилом «молодший байт за молодшою адресою».

Дещо специфічним є тип даних, що визначається за допомогою директиви DT. Його основне призначення – підтримка операцій двійково-десяткової арифметики (також використовується для обміну даними між основним процесором і співпроцесором). Дані цього типу не поділяються на будь які інші структурні одиниці, і являють собою масив із 10 байтів, але в пам'яті, на відміну від «справжніх» масивів байт, усі елементи такого масиву також розміщуються як little-endian.

Таке перекидання «*молодший байт за молодшою адресою*» під час запису в пам'ять машинних слів *мікропроцесор виконує автоматично*. Під час читання з пам'яті мікропроцесор *автоматично* виконує зворотне перетворення. Наприклад, якщо дані визначені так:

```
.data
x       dq      1020304050607080h
y       dd      10203040h
z       dw      1020h
u       db      10h
v       db      20h
```

то в пам'яті вони матимуть такий вигляд:

```
xxxxxxxxxxxxxx00h       80 70 60 50 40 30 20 10
xxxxxxxxxxxxxx08h       40 30 20 10
xxxxxxxxxxxxxx0Ch       20 10
xxxxxxxxxxxxxx0Eh       10
xxxxxxxxxxxxxx0Fh       20
```

Числа xxxxxxxxxxxxxx00h … xxxxxxxxxxxxxx0Fh – це адреси, що ставляться у відповідність символічним іменам змінних x, y, z, u та v. Символи «xxxxxxxxxxxxxx» позначають частину адреси, яка буде відома після

завантаження програми на виконання. Наведена структура (дамп) пам'яті наглядна, а для мікропроцесора пам'ять лінійна і має вигляд послідовності байт.

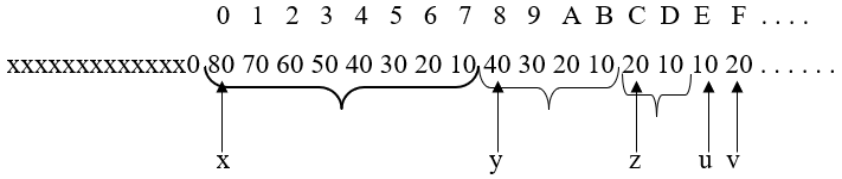


Рисунок 2.4 – Послідовність байт у пам'яті

На рисунку 2.4 xxxxxxxxxxxxx0 позначає адресу, із якої починається розміщення даних секції. Цю адресу називають базовою. Символи 0, 1, … F над значеннями байтів позначають їх зміщення відносно базової адреси.

Мікропроцесор оперує не символічними іменами змінних, а числами. Загалом *символічне ім'я об'єкта програми (змінна, мітка, підпрограма) являє собою число, значення якого дорівнює адресі комірки пам'яті, із якої починається розміщення об'єкта, позначеного цим символічним ім'ям*.

Отже, якщо символічні імена, визначені у секціях даних або коду інтерпретуються асемблером як числа, то їх можна використовувати як значення ініціалізації:

ім'я Директива <символічне ім'я об'єкта програми>

Директива для 64-розрядного може бути DD або DQ. У пам'яті буде розміщено значення адреси, що відповідає <символічне ім'я об'єкта програми>. Інакше кажучи, використання такої директиви дозволяє отримати адресу певного об'єкта програми за його символічним ім'ям. Наприклад, якщо дані визначені так:

```
.data
x    dd    10203040h
y    dq    1020304050607080h
z    dq    y,
```

то в пам'яті вони матимуть такий вигляд:

xxxxxxxxxxxxxx00h 40 30 20 10

xxxxxxxxxxxxxx04h 80 70 60 50 40 30 20 10

xxxxxxxxxxxxxx0Ch 04 xx xx xx xx xx xx xx

2.3 Склад, розрядність і призначення регістрів

Рисунок 2.5 відображає склад і розрядність регістрів Intel64/AMD64. Слід зазначити що в підрежимі сумісності режиму IA-32e, у реальному та захищеному режимі склад і розрядність регістрів Intel64/AMD64 збігається з IA-32.

У процесі розробки архітектури Intel64/AMD64 вона успадкувала від IA-32 частину регістрів. Вісім 32-розрядних регістрів загального призначення з іменами EAX, EBX, ECX, EDX, EBP, ESI, EDI, ESP перетворилися на 64-розрядні регістри RAX, RBX, RCX, RDX, RBP, RSI, RDI, RSP, і було ще додано вісім рігістрів R8–R15. Розрядність регістру вказівника команд EIP і регістру стану та керування EFLAGS зільшилася з 32-х до 64-х бітів.

Регістри FPU, MMX розглядатимуться згодом, а вивчення інших регістрів почнемо з регістрів загального призначення.

2.3.1 Регістри загального призначення

Рисунок 2.6 відображає структуру регістрів загального призначення (РЗП) Intel64/AMD64. Схематичне подання регістрів загального призначення на рисунку вимагає деяких пояснень. По-перше, на рисунку відображені *символічні імена регістрів, які використовуються в як операнди машинних команд* під час запису їхніх мнемонічних позначень у програмах, а по-друге, рисунок відбиває також і внутрішню структуру регістрів.

Мікропроцесор Intel64/AMD64 має шістнадцять 64-розрядних регістрів з іменами RAX, RBX, RCX, RDX, RBP, RSI, RDI, RSP, R8–R15. Молодші 32 біта (31–0) кожного із цих регістрів мають власні імена EAX, EBX, ECX, EDX, EBP, ESI, EDI, ESP, R8D–R15D і можуть використовуватися як самостійні регістри, а

старші половини (біти 63–32) 64-х розрядних регістрів не мають імен і як самостійна одиниця даних недоступні.

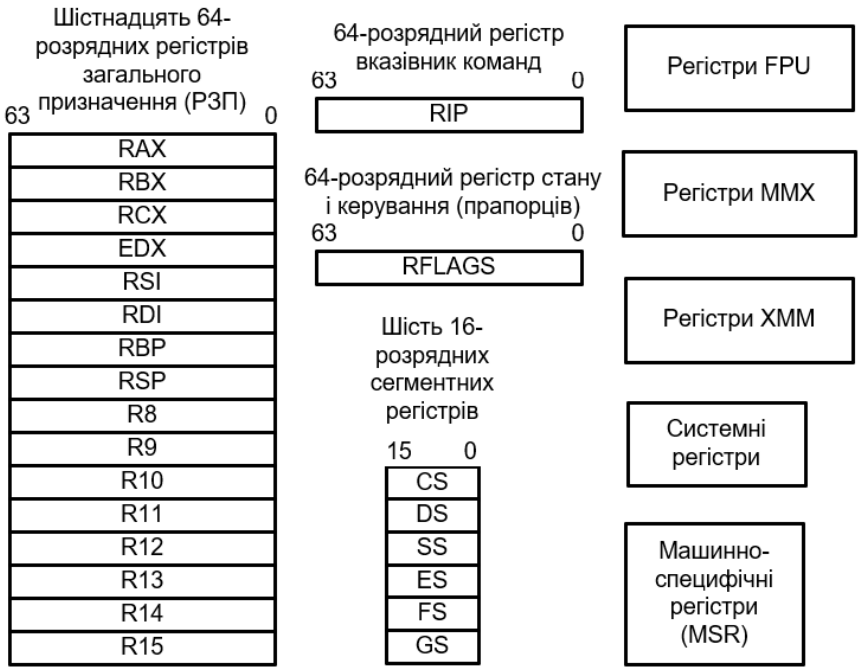


Рисунок 2.5 – Склад і розрядність регістрів Intel64/AMD64

Молодші 16 бітів (15–0) кожного з регістрів RAX, RBX, RCX, RDX, RBP, RSI, RDI, RSP, R8–R15 (або молодші 16 бітів кожного з регістрів EAX, EBX, ECX, EDX, EBP, ESI, EDI, ESP, R8D–R15D) мають власні імена AX, BX, CX, DX, BP, SI, DI, SP, R8W–R15W і можуть використовуватися як самостійні регістри.

Молодші 8 бітів (7–0) кожного з регістрів RAX, RBX, RCX, RDX, RBP, RSI, RDI, RSP, R8–R15 (молодші 8 бітів кожного з регістрів EAX(AX), EBX(BX), ECX(CX), EDX(DX), EBP(BP), ESI(SI), EDI(DI), ESP(SP), R8D(R8W) – R15D(R15W) також мають власні імена, що надає ще шістнадцять байтових регістрів AL, BL, CL, DL, BPL, SIL, DIL, SPL, R8B–R15B.

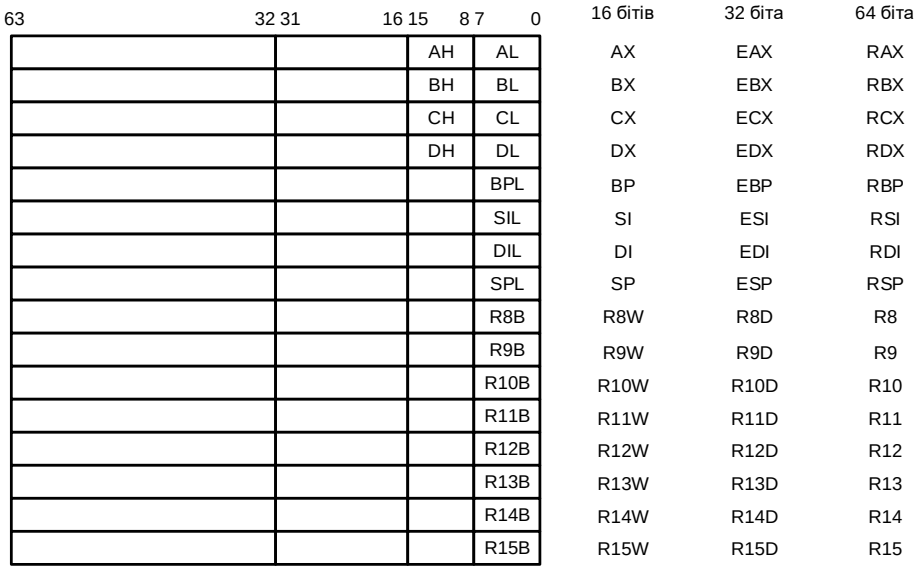


Рисунок 2.6 – Структура регістрів загального призначення Intel64/AMD64

Окрім цього, старші (15-8) біти регістрів AX, BX, CX, DX також мають власні імена, що надає ще чотири байтових регістри AH, BH, CH і DH, які можуть використовуватися як окремий апаратний ресурс для маніпуляцій з байтами даних.

Отже, кожний з регістрів RBP, RSI, RDI, RSP, R8–R15 містить три регістри – один байтовий, один двобайтовий і один чотирибайтовий, а кожний з регістрів RAX, RBX, RCX, RDX містить чотири регістри – два байтові, один двобайтовий і один чотирибайтовий, що надає можливість використання їх для обробки різних апаратно підтримуваних типів даних. Особливості використання їхньої структури в програмах на асемблері такі.

1. Старші половини 32- та 64-розрядних регістрів як самостійна одиниця даних недоступні і доступ до них безпосередньо, тобто однією командою, можливий тільки у складі 32- або 64-розрядного регістра.

2. Старші половини 16-розрядних регістрів BP, SI, DI, SP, R8W–R15W як самостійна одиниця даних недоступні і доступ до них безпосередньо, тобто однією командою, можливий тільки у складі 16-розрядного регістра.

3. Будь-які зміни бітів 63–32 змінюють значення 64-розрядного регістра в цілому, але не змінюють значення регістра, що являє собою його 32 молодших біта.

4. Будь які зміни бітів 31–16 змінюють значення 32-розрядного регістра в цілому, але не змінюють значення регістра, що являє собою його 16 молодших бітів.

5. Будь які зміни старшої половини (біти 15–8 або регістри AH, BH, CH, DH) 16-розрядних регістрів AX, BX, CX, DX не змінюють значення його молодшої частини (біти 7–0 або регістри AL, BL, CL, DL), і навпаки, але змінюють значення всього 16-розрядного регістра, і, звичайно, у цілому всього 32- та 64-розрядного регістру.

2.3.2 Сегментні регістри

Сегментні регістри CS, DS, SS, ES, FS, GS відіграють значну роль в під-режимі сумісності режиму IA-32e, у захищеному і реальному режимі. Вони являють собою той апаратний ресурс, що використовується для розподілу й керування пам'яттю. Значення в сегментному регістрі називаєть *селектор,* і воно *визначає* базову адресу секції, і в якій власне секції (коду, стека, даних) перебуває об'єкт у пам'яті. Під час виділення пам'яті програмі операційна система завантажує значення селекторів у відповідні сегментні регістри: у CS – для секції коду, у DS – для секції даних, у SS – для секції стека, у ES, FS й GS – для додаткових і спеціальних секцій.

У long mode регістри CS, DS, SS, ES не використовуються і задіяні тільки FS і GS. Сегментні регістри FS і GS у long-mode використовуються операційними системами для адресації власних структур даних. Зокрема 64-розрядні Windows

NT використовують GS для доступу до системної структури даних Thread Environment Block (TEB).

2.3.3 Регістр-покажчик команд і стекові регістри

Для адресації команд використовується регістр RIP (Instruction Pointer Register) – регістр-покажчик команд. Завжди, у будь-який час, RIP містить адресу машинної команди програми, що підлягає виконанню мікропроцесором. Початкове завантаження RIP під час старту програми виконується засобами операційної системи. Початкове значення RIP визначає розробник програми, а подальші зміни RIP автоматично виконує сам мікропроцесор після виконання кожної машинної команди.

Стек – це пам'ять типу LIFO (Last Input First Output, тобто останнім прийшов – першим пішов). У стековій пам'яті осередки утворюють одномірний масив, операції запису/читання елементів якого виконуються через так звану верхівку стека.

Завдяки з важливості структури стекової пам'яті для організації обчислювального процесу, вона в більшості процесорів підтримується на програмно-апаратному рівні. За такої реалізації стек розміщується в основній, тобто адресній пам'яті. На верхівку стека постійно вказує значення регістра покажчика стека процесора, яке автоматично змінюється самим процесором унаслідок виконання операцій читання/запису даних стеку.

Доступ процесора до даних стека здійснюється як до пам'яті з довільним доступом, а для програмного забезпечення з використанням *спеціалізованих команд для роботи зі стеком* має вигляд, як доступ до пам'яті з послідовним доступом.

Апаратну підтримку структури стека в мікропроцесорах Intel64/AMD64 забезпечує регістр RSP (Stack Pointer Register) – регістр-покажчик верхівки стека. Завжди, у будь-який час, RSP містить поточну адресу верхівки стека, за якою команди для роботи зі стеком виконують читання і запис даних. Початкове

значення RSP встановлюється операційною системою під час старту програми, а подальші зміни RSP автоматично виконуються самими мікропроцесором у результаті виконання спеціалізованих команд для роботи зі стеком і деяких інших.

Із підтримкою структури стека також пов'язаний регістр RBP − регістр покажчика бази кадру стека. Регістр RBP (Base Pointer Register) традиційно має вузько спеціалізоване призначення. Він використовується для довільного доступу (зазвичай неруйнівне читання) до будь-якого елемента в глибині стека.

Регістри RSP і RBP програмно доступні, але не слід змінювати їх значення *без особливої потреби*. Навіть якщо дійсно така необхідність виникає, то до їх модифікації варто підходити дуже відповідально, бо некоректні дії стосовно цієї пари регістрів можуть призвести до руйнування структури стека, і в наслідок цього − можливого подальшого краху програми.

Незважаючи на дещо особливий статус стека, він нічим не відрізняється від звичайних даних. Принцип LIFO стосується лише *спеціалізованих команд* для роботи зі стеком, а в інших випадках можна вважати, що стек – це звичайна пам'ять, до якої можна звертатися за адресою, яка визначається регістром RSP.

На відміну від коду, доступ до якого виконується послідовно, команда за командою, і стека, доступ до якого виконується згідно з принципом LIFO, *доступ до даних виконується довільно*, тобто мікропроцесор не може знати, до яких даних буде звернення наступного моменту, отже і спеціально визначеного регістра для зберігання адреси «наступних» даних немає.

Призначення інших регістрів або груп регістрів визначимо пізніше при розглядаючи систему команд процесора. Поки що вважаємо, що кожен з РЗП ми можна використовувати коли завгодно і як завгодно, що з огляду на мікропроцесор стосовно РЗП цілком правильно, але на практиці може призвести до логічних помилок у програмуванні.

2.3.4 Регістр стану та керування

Регістр RFLAGS є впорядкованою послідовністю ознак або прапорців, що визначають *стан обчислюваного процесу в кожен момент часу*. Значення прапорців формується автоматично внаслідок виконання команд, частина прапорців може встановлюватися програмно, тобто відповідною командою. Стан деяких прапорців RFLAGS впливає на поведінку процесора під час виконання певних команд.

Під час опису алгоритмів машинних команд обов'язково зазначається, як команда впливає на стан регістру RFLAGS та/або як стан RFLAGS впливає на поведінку процесора внаслідок виконання команд. Найбільш «популярні» прапорці зазначено в таблиці 2.2.

Таблиця 2.2 – Призначення RFLAGS (вибірково)

Прапор	Біт	Зміст і призначення
CF (Carry Flag) Прапор перенесення	0	1 – арифметична операція зробила перенесення зі старшого (знакового) біта результату; 0 – перенесення не було
ZF (Zero Flag) Прапор нуля	6	1 – результат операції нульовий, 0 – ненульовий.
SF (Sign Flag) Прапор знака	7	Установлюється рівним старшому (знаковому) біту результату операції
OF (Overflow Flag) Прапор переповнення	11	1 – у результаті операції відбувається перенесення в старший (знаковий) біт результату, або позика з нього, 0 – не відбувається.

2.4 Способи адресації операндів машинних команд

2.4.1 Загальна структура машинної команди

Машинна команда являє собою закодовану за певними правилами послідовність байт, що містить в узагальненому випадку вказівку мікропроцесору про вид виконуваної операції, місце розташування операндів і результату. Спрощену структуру машинної команди представлено рис. 2.7.

Операційна частина	Адресна частина

Рисунок 2.7 – Загальна структура машинної команди

Операційна частина містить код операції (КОП), що визначає вид елементарних машинних дій (додавання, логічне множення, пересилання і.т. д.), які виконуватиме мікропроцесор, а адресна частина містить операнди, або в загальному випадку інформацію про місце розташування їх та інформацію про місце розташування результату.

Під час запису програми на асемблері використовуються мнемонічні позначення (мнемоніки) машинних команд із зазначенням набору операндів кожної команди. Для відокремлення одне від одного операндів машинної команди використовується символ «,» (кома). Мнемоніка визначає КОП, а набір операндів – адресну частину.

Машинні команди можуть зовсім не мати операндів, мати один, два і три операнди:

```
Мнемоніка
Мнемоніка       операнд
Мнемоніка       операнд1 , операнд2
Мнемоніка       операнд1 , операнд2 , операнд3
```

Команди бінарних (тобто тих, що мають два операнди) операцій називають *командами основної групи*.

Для позначення мнемонік і набору операндів команд у MASM прийнятий синтаксис Intel – система позначень, яка використовується самим розробником мікропроцесорів у офіційній технічній документації. У синтаксисі позначень Intel ***лівий*** *(за деякими винятками) операнд є приймачем*, тобто в ньому внаслідок виконання команди буде розміщено результат. Для команд основної групи

Мнемоніка операнд-приймач, операнд-джерело

алгоритм виконання полягає в тому, що:

операнд-приймач <= операнд-приймач ○ операнд-джерело

Виконується операція, що визначається КОП і позначена символом «○» над операнд-приймач і операнд-джерело і результат розміщується в операнд-приймач.

2.4.2 Місцезнаходження операндів машинних команд

Операнди машинних команд можуть бути розташовані:

1. У місці, що *неявно визначається кодом операції* команди.
2. Безпосередньо в коді машинної команди.
3. У регістрах загального призначення.
4. У пам’яті.
5. У стеку (пам’яті).
6. У портах вводу/виводу.
7. У стеку арифметичного співпроцесору.
8. У MMX-регістрах.
9. У XMM-регістрах.

Видається неможливим розглядати способи задання й адресацію операндів машинних команд окремо від самих машинних команд. Розглянемо команду MOV, на прикладі якої можна продемонструвати більшість способів адресації операндів машинних команд.

Мнемоніка	MOV приймач, джерело

Призначення	Пересилання даних між регістрами або регістрами й пам’яттю, завантаження в регістри або в пам’ять безпосередніх значень
Алгоритм	Виконує *копіювання* значення операнда «джерело» в операнд «приймач»
RFLAGS	*У переважній більшості випадків* не впливає

Повний опис операндів команди MOV, так само як і деяких інших, займатиме не одну сторінку. За повним описом команд краще звернутися до оригінальної документації Intel® 64 and IA-32 Architectures Software Developer's Manuals (http://www.intel.com/products/processor/manuals/) або скористатися наведеним списком літератури.

Можна сформулювати три прості правила (***до яких існують винятки***), дотримання яких позбавить певною мірою від синтаксичних помилок у командах основної групи.

1. Операнди повинні бути однакового розміру (у деяких випадках лівий операнд повинен мати не менший розміру, ніж правий).
2. Обидва операнди не можуть розміщуватися в пам’яті.
3. Лівий операнд не може задаватися безпосереднім значенням (константою, числом).

2.4.3 Регістрова й безпосередня адресація

Регістрова й безпосередня адресація являють собою найпростіші та інтуїтивно зрозумілі способи адресації, які й адресацією назвати важко. У разі безпосередньої адресації операнд задається безпосереднім значенням (тобто числом/константою), або виразом, побудованим з використанням символьних імен інших об’єктів програми і операторів асемблера.

Транслятор повинен мати можливість обчислити вираз на етапі трансляції, після чого він також буде числом/константою. Для зберігання безпосереднього

операнда в адресній частині коду команди виділяється поле відповідної довжини, тому такі команди за довгих операндів мають довгий машинний код.

У випадку регістрової адресації операнд в адресній частині машинного коду команди задається ім'ям (кодом, адресою) регістра, у якому він розташований. Наприклад:

mov al,bl – обидва операнди регістрові;

mov al,100 – регістр і безпосереднє значення;

mov ax,si – обидва операнди регістрові;

mov ax, 101010101010101b – регістр і безпосереднє значення;

mov eax,esi – обидва операнди регістрові;

mov eax, 0BE2F70h – регістр і безпосереднє значення;

mov rcx, rbx – обидва операнди регістрові;

mov rax, -1 – регістр і безпосереднє значення.

Адресація операндів у пам'яті є більш складною.

2.4.4 Адресація операндів у пам'яті

Адреса операнда у пам'яті у довгому режимі Intel64/AMD64 зазначається у вигляді виразу $[\ Base + Index * Scale + Displacement\ ]$, у якому квадратні дужки «[ ]» позначають спеціальний адресний операнд, а символи «+» та «*» є елементами синтаксису. Рисунок 2.8 відображає можливі значення елементів виразу Base, Index, Scale та Displacement.

$$Base = \begin{cases} RAX \\ RBX \\ RCX \\ RDX \\ RSP \\ RBP \\ RSI \\ RDI \\ R8 \\ ... \\ R15 \end{cases} \quad Index = \begin{cases} RAX \\ RBX \\ RCX \\ RDX \\ RBP \\ RSI \\ RDI \\ R8 \\ ... \\ R15 \end{cases} \quad Scale = \begin{cases} 1 \\ 2 \\ 4 \\ 8 \end{cases} \quad \text{Displacement} = \begin{cases} imm8 \\ imm16 \\ imm32 \end{cases}$$

Рисунок 2.7 – Можливі значення елементів виразу Base, Index, Scale та Displacement

Отже, усі способи адресації операндів у пам'яті є окремими випадками загальної схеми $[Base + Index * Scale + Displacement]$. Будь яка адреса формується вибором відповідного елемента в множинах «Base», «Index», «Scale» і «Displacement»:

1. Як Base (базовий регістр) можуть використовуватися усі 64-розрядні регістри загального призначення, а як Index (індексний регістр) – усі 64-розрядні РЗП, окрім RSP.

2. Масштабний коефіцієнт Scale може мати одне зі значень з множини {1, 2, 4, 8}.

3. Зміщення Displacement може бути 8/16/32-розрядним числом зі знаком.

Додаткові умови:

1. Як Base (базовий регістр) та Index (індексний регістр) можуть використовуватися 32-розрядні регістри EAX, EBX, ECX, EDX, ESI, EDI, ESP (тільки для Base), EBP, R8D-R15D.

2. Одночасне (змішане) використання базового (Base) та індексного (Index) 32-розрядних і 64-розрядних регістрів неприпустимо.

3. Можливе використання адресації *[RIP + Displacement]* із 32-розрядним Displacement, яке внаслідок додавання до RIP знаково розширюється до 64 біт.

4. Якщо RSP або RBP використовуються як базовий регістр, то виконується адресація стекової пам’яті.

2.4.4.1 Базова адресація

За такого типу адресації адреса операнда знаходиться в регістрі загального призначення. Для синтаксичного позначення такого типу адресації використовується адресний оператор [], тобто ім’я базового регістра (Base) розміщується в квадратних дужках. Наприклад, команда mov ax,[rbx] виконує пересилання в регістр AX слова з адреси, що дорівнює числу у регістрі RBX.

2.4.4.2 Базова адресація зі зміщенням

Цей вид адресації є розширенням базової адресації. До значення, що міститься в базовому регістрі, можна додавати/віднімати зміщення (Displacement).

Синтаксично цей спосіб адресації позначається символом «+/-». Наприклад, команда mov ax,[rbx+1] виконує пересилання в регістр AX слова з адреси, що на одиницю більша, ніж число в регістрі RBX.

2.4.4.3 Індексна адресація з масштабуванням

Синтаксично цей спосіб адресації позначається множенням індексного регістра (Index) на масштабний коефіцієнт (Scale), що дорівнює 1, 2, 4 або 8 (відповідно до розміру в байтах апаратно підтримуваних типів даних). Індексним регістром може бути будь-який 32/64-розрядний регістр загального призначення, окрім ESP і RSP. Наприклад, команда mov ax,[rbx*2] виконує

пересилання в регістр AX слова, із адреси в 2 рази більшої, ніж число в регістрі RBX.

2.4.4.4 Індексна адресація з масштабуванням і зміщенням

Цей вид адресації є розширенням адресації з масштабуванням. До значення, що розміщується в індексному регістрі (Index), після множення його на масштабний коефіцієнт можна додавати ще й зміщення (Displacement). Синтаксично цей спосіб адресації також позначається символом «+/-».

Наприклад, команда mov eax,[rbx*2+1] виконує пересилання в регістр EAX слова з адреси на 1 більше, ніж подвоєне число, що розміщується в регістрі RBX.

2.4.4.5 Базово-індексна адресація

Цей тип адресації схожий на базову адресацію зі зміщенням, якщо вважати що значення зміщення вже задається не числом, а ім’ям регістра загального призначення. Синтаксично цей спосіб адресації позначається «+» між іменами базового й індексного регістра. Можливими комбінаціями в парах Base+Index можна вибирати будь-який з 32/64-розрядних РЗП крім пар ESP+ESP і RSP+RSP.

Наприклад, команда mov ax,[rbx+rsi] виконує пересилання в регістр AX, слова з адреси, що дорівнює сумі чисел у регістрах RBX і RSI.

2.4.4.6 Базово-індексна адресація зі зміщенням

Цей тип адресації є розширенням базово-індексної адресації, тепер до пари Base+Index ще можна додати й зміщення (Displacement). На прикладі цього типу адресації продемонструємо різні синтаксичні форми запису адреси. Наведені нижче 4 команди еквівалентні:

```
mov rax,[rbx+rdi+2]
mov rax,[rbx][rdi+2]
mov rax,[rbx+2][rdi]
mov rax,2[rbx][rdi]
```

Унаслідок обчислення адреси буде виконано додавання значень регістрів RBX і RDI, а до результату ще буде додано 2.

2.4.4.7 Базово-індексна адресація з масштабуванням

Цей вид адресації є черговим розширенням базово-індексної адресації. Такі команди еквівалентні:

```
mov rax,[rbx+rdi*2]
mov rax,[rbx][rdi*2]
mov rax,[rdi*2][rbx].
```

Унаслідок обчислення адреси буде виконано додавання значення регістра RBX і подвоєного значення регістра RDI.

2.4.4.8 Базово-індексна адресація з масштабуванням і зміщенням

Це найповніша схема адресації, у якій задіяні всі складові Base, Index, Scale і Displacement. Різні синтаксичні форми запису адреси:

```
mov rax,[rbx+rdi*2+1]
mov rax,[rbx][rdi*2+1]
mov rax,[rbx+1][rdi*2]
mov rax,1[rbx][rdi*2]
mov rax,1[rdi*2][rbx].
```

Унаслідок обчислення адреси буде виконане додавання значення регістра RBX і подвоєного значення регістра RDI, а потім до результату ще додано 1.

2.4.4.9 Пряма адресація

Це найпростіший тип адресації операнда в пам’яті, оскільки що адреса розміщується безпосередньо в машинному коді команди, і ніяких обчислень мікропроцесору виконувати непотрібно.

Така форма прямої адресації використовується тільки в реальному режимі роботи мікропроцесора за необхідності звернутися безпосередньо за певною

фізичною адресою пам'яті, і для цього *явно повинен зазначатися сегментний регістр*.

Для позначення адреси пам’яті також використовується індексний оператор асемблера квадратні дужки «[]», наприклад, mov ax, ds:[0000h] або mov bx, es:[0100h].

Зазвичай всім областям пам’яті, які використовує програма, привласнюються символічні імена, якими оперує програма. Зокрема це змінні, визначені в секціях даних та/або констант.

Використання в програмах символічних імен змінних – це завуальована форма адресації. *У разі використання символічних імен змінних транслятор «бачить» у якій секції вони визначені, і їх тип*, що дозволяє йому обчислити у процесі трансляції значення адрес, які відповідають змінним програми з урахуванням їх типу, і сформувати машинний код команди.

2.4.5 Оператор уточнення типу покажчика на пам’ять

У деяких випадках під час адресації пам’яті можливо неоднозначне трактування команди, наприклад: як транслятор має розуміти команду mov [rsi], 5, скільки байтів розміщувати в пам’яті за адресою в регістрі RSI з урахуванням того, що константа 5 може бути подана як байтом, так і словом або подвійним чи подвійним подвійним словом?

Для уточнення типу операндів у пам’яті використовується оператор PTR, який має вигляд <тип> PTR <операнд>. Вираз <тип> може зазначатися синонімами (Таблиця 2.1), які відповідають директивам ініціалізації та резервування пам’яті, або зазначатися іменем типу, визначеного за допомогою TYPEDEF.

Асемблер зв’язує з кожним символічним іменем атрибут типу, що дозволяє йому контролювати апаратну сумісність даних, а щодо логічної інтерпретації – йому байдуже. Якщо операнд задається символічним іменем, то транслятор,

ураховуючи атрибут типу операнда, може видавати попередження про невідповідність розмірів операндів.

Наприклад, команди mov eax,x та mov x,eax будуть правильними за умови, якщо змінна x визначена за допомогою директив DD/DWORD/SDWORD/REAL4. Як діяти, якщо потрібно обробити, наприклад, молодший байт або молодше слово цієї змінної, а транслятор не сприймає команди mov al,x та mov ax,x попереджаючи про невідповідність типів?

Допоможе оператор PTR, який, за необхідності, також можна використовувати для перевизначення типу операндів у пам'яті, і «домовитися» з асемблером можна, записавши попередні команди як mov al,byte ptr x та mov ax,word ptr x.

Символічне ім'я змінної являє собою число, значення якого дорівнює адресі комірки пам'яті, у якій починається розміщення змінної. До адреси пам'яті, тобто до імені змінної, можна додавати або віднімати від неї інші числа, і, використовуючи оператор PTR, можна отримати доступ до будь якої складової частини багатобайтових даних.

Приклад. У секціях даних визначені такі змінні:

```
.data
X1   dd   11223344h
X2   dd   55667788h
.data?
Y1   dw   ?
Y2   dw   ?
```

За допомогою команд MOV отримати в зазначених комірках пам'яті такі значення: 8811h в Y1, 2233h в Y2, 77881122h в Z, де 11h, 22h, 33h і т. д. Це відповідні байти змінних X1 та X2.

Дамп пам'яті для змінних X1 та X2 матиме такий вигляд:

```
        0  1  2  3  4  5  6  7
x00   44 33 22 11 88 77 66 55
```

X1 X2

Отже байти, що складають слова 8811h, 2233h і подвійне слово 77881122h хоча і належать різним змінним, але в пам’яті розміщуються поряд. Тому адресувати потрібні дані можна зі зміщенням (позитивним чи негативним) відносно адрес X1 або X2.

Оскільки пересилання з пам’яті в пам’ять командою MOV виконати неможливо, то операцію потрібно виконувати за допомогою буферного регістру. Реалізувати це можна, наприклад, таким (*пам’ятаємо правило «молодший байт за молодшою адресою»*):

```
mov ax,word ptr X1+3
mov Y1,ax
mov ax,word ptr X1+1
mov Y2,ax
mov eax, dword ptr X1+2
mov Z,eax
```

2.5 Система команд процесора

Під час опису синтаксису машинних команд використовуються спеціальні позначення для операндів:

- imm – безпосередній операнд (число);
- r – будь-який регістр загального призначення;
- m – операнд у пам’яті;
- Sreg – сегментний регістр;
- rel – відносний адресний операнд;
- moffs – змінна простого типу.

Опис синтаксису машинних команд часто вимагає визначення розмірності операндів, і у цьому випадку використовуються такі позначення:

- imm8, imm16, imm32, imm64 – безпосередній операнд, для розміщення якого потрібно 8/16/32/64 біта;

– r8 – AH, AL, BH, BL, CH, CL, DH, DL, BPL, SPL, DIL, SIL, R8L–R15L;

– r16 – AX, BX, CX, DX, BP, SP, SI, DI, R8W–R15W;

– r32 – EAX, EBX, ECX, EDX, EBP, ESP, ESI, EDI, R8D–R15D;

– r64 – RAX, RBX, RCX, RDX, RBP, RSP, RSI, RDI, R8–R15;

– ac – акумулятор AL, AX, EAX, RAX (окрім деяких випадків це просто r або r8, r16, r32, r64);

– m8, m16, m32, m64, m128 – операнд заданого розміру у пам'яті;

– rel8/rel16/rel32/rel64 – відносний адресний операнд заданого розміру;

– moffs8/moffs16/moffs32/moffs64 – змінна простого типу заданого розміру;

– m16:16, m16:32, m16:64 – покажчики на пам’ять, розташовані у пам'яті

– m16&32, m16&16, m32&32, m16&64 – операнд у пам’яті, що складається з окремих частин заданого розміру.

Для вивчення системи команд процесора їх зручно розділити на декілька груп (Рисунок 2.8) і розглядати окремо за групами.

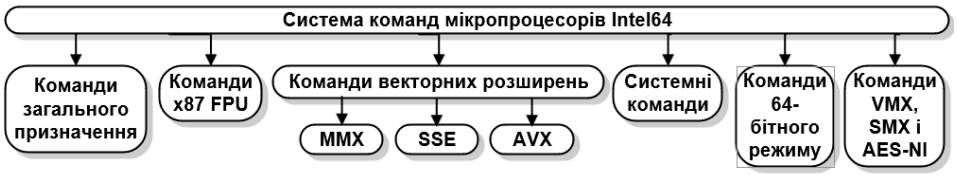


Рисунок 2.8 – Система команд процесора

Кожна наступна група команд, що зображені на рисунку 2.9, у порядку зліва направо розширюватиме наші можливості щодо програмування процесора.

2.5.1 Команди загального призначення

Рисунок 2.9 – Команди загального призначення

2.5.1.1 Команди переміщення даних

У групі команд переміщення можна виокремити декілька підгруп

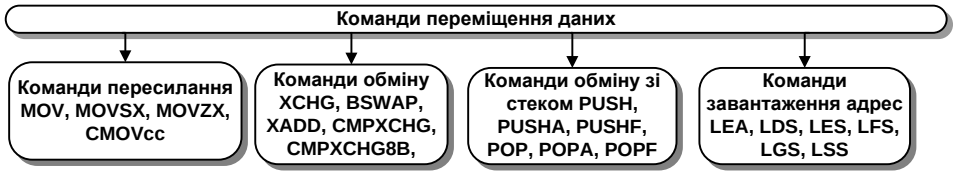


Рисунок 2.10 – Команди переміщення

Ця група команд надзвичайно важлива, оскільки робота з мікропроцесором без більшої частини команд цієї групи стає неможливою. Ці команди дозволяють завантажувати операнди з пам’яті для їх подальшої обробки і зберігати результати в пам’яті, переміщувати значення поміж регістрами та пам’яттю, працювати зі стеком і адресами.

2.5.1.1.1 Команди пересилання

Про основну команду пересилання MOV ми йшлося раніше під час розгляду способів адресації операндів машинних команд. Ознайомлення з командами MOVSX і MOVZX логічніше відкласти до розгляду групи арифметичних команд, а ознайомлення із командою CMPXCHG і групою команд CMOVcc – до розгляду команд передавання керування.

2.5.1.1.2 Команди обміну

Мнемоніка XCHG <операнд1>,<операнд2>

Призначення	Обмін вмісту регістрів, регістрів і комірок пам'яті
Алгоритм	На місці <операнд1> розміщується <операнд2>, а на місці <операнд2> розміщується <операнд1>
Операнди	r–r, r–m, m –r
RFLAGS	не впливає

Приклад: Обмін значеннями двох змінних

```
.data
X    db    10h
Y    db    20h
```

1)
```
mov al,X
mov ah,Y
mov X,ah
mov Y,al
```

2)
```
mov ax,word ptr X
xchg  al, ah
mov word ptr X, ax
```

3)
```
mov  al,X
xchg  al, Y
xchg  al, X
```

У першому випадку використовується чотири команди і чотири звертання до пам'яті, незалежно від розташування операндів у пам'яті.

У другому випадку задачу розв'язано за три команди з двома звертаннями до пам'яті, користуючись тим, що операнди розміщено у сусідніх комірках.

У третьому випадку використовується три команди з трьома звертаннями до пам'яті, **незалежно** від розташування операндів у пам'яті.

Мнемоніка BSWAP <операнд>

Призначення	Перетворення подання little-endian у big-endian
Алгоритм	Обертає байти <операнд> у зворотному порядку байтів
Операнди	r32, r64
RFLAGS	не впливає

Приклад. Подання чисел у little-endian та big-endian, доступ до старшої половини регістра.

```
mov eax,11223344h          ; завантажити в регістр EAX число
mov dword ptr [rsi], eax   ; розміщення числа в пам’яті як little-endian
bswap eax                  ; у регістрі EAX буде 44332211h
; розміщення числа в пам’яті як big-endian
mov dword ptr [rsi], eax
mov ax, 6655h              ; у регістрі AX буде 6655h
bswap eax                  ; у регістрі EAX буде 5566xxxxh
mov ax,7788h               ; у регістрі EAX буде 55667788h
; скласти в EDX подвійне слово з  DX:AX
mov dx,8877h               ; у регістрі DX буде 8877h
mov ax,6655h               ; у регістрі AX буде 6655h
; перевернути байти в DX для того щоб перемістити їх у старшу половину EDX
xchg dh,dl                 ; у регістрі DX буде 7788h
; bswap знову переверне байти
bswap edx                  ; у регістрі EDX буде 8877xxxxh
mov dx,ax                  ; у регістрі EDX буде 88776655h
```

2.5.1.1.3 Команди обміну зі стеком

Безпосереднє призначення стека:

1. Збереження стану і адреси повернення в програму, що переривається для обробки апаратних і програмних переривань.

2. Збереження адреси повернення в програму, яка викликає інші підпрограми.

3. Передавання параметрів у підпрограми.

4. Тимчасове зберігання локальних змінних підпрограм.

5. Тимчасове зберігання проміжних результатів програм/підпрограм.

Команди обміну зі стеком:

Мнемоніка PUSH <операнд>

Призначення	Завантаження у стек операнда
Алгоритм	Залежно від розміру операнда зменшує значення регістра-покажчика верхівки стека RSP на 2/8 і копіює операнд у пам’ять за адресою в регістрі-покажчику верхівки стека RSP.
Операнд	imm8/imm16/imm32/r16/r64/m16/ m64/Sreg (тільки FS і GS)
RFLAGS	Не впливає

Мнемоніка POP операнд

Призначення	Завантаження в операнд даних на верхівці стека
Алгоритм	Залежно від розміру операнда копіює 2/8 байтів з пам’яті за адресою в регістрі-покажчику верхівки стека RSP в операнд і збільшує значення регістра-покажчика верхівки стека на 2/8.
Операнд	r16/m16/r64/m64/Sreg (Sreg тільки FS і GS)
RFLAGS	Не впливає

Мнемоніка PUSHF/PUSHFQ/POPF/POPFQ

Призначення	Збереження/відновлення в/зі стека регістра прапорців
Алгоритм	PUSHF/PUSHFQ зменшує значення регістра-покажчика верхівки стека RSP на 4/8 і копіює значення регістра прапорців у пам’ять за адресою в регістрі-покажчику верхівки стека, а POPF/POPFQ виконує зворотні дії
Операнд	Немає
RFLAGS	Не впливає

За допомогою команд обміну зі стеком можуть бути виконані й деякі інші операції, для виконання яких структура стека безпосередньо не призначена. Наприклад:

1) пересилання і обмін пам’ять–пам’ять

```
……
X     QWORD    1
Y     QWORD    2
……
push  X
pop   Y
push  X
push  Y
pop   X
pop   Y
```

2) доступ до регістра RFLAGS

```
pushfq          mov rax, 1
pop rax         push rax
                popfq
```

2.5.1.1.4 Команди завантаження адрес

Мнемоніка LEA <приймач>, <джерело>

Призначення	Завантаження в регістр адреси пам’яті, за якою розташовуються дані простих чи структурованих типів для подальшої їх обробки з використанням базової/базово-індексної адресації
Алгоритм	Завантажує в <приймач> адресу <джерела>
Операнд	Приймач – r16/r32/r64, джерело – m
RFLAGS	Не впливає

Приклад:

```
.data
X           BYTE    1
Y           WORD    2
Z           DWORD   3
Array       DWORD   4, 5, 6
……
lea rbx, X
mov al, byte ptr [rbx]              ; AL = 1
mov ax, word ptr [rbx+1]            ; AX = 2
mov eax, dword ptr [rbx+3]          ; EAX = 3
lea rbx, Array
mov rsi,1
mov eax, dword ptr [rbx+rsi*4]      ; EAX = 5
```

Команду LEA, окрім її прямого призначення, можна використовувати для виконання арифметичних операцій. Приклад:

1. Триоперандне додавання:

```
lea rcx, [rax+rbx]  ; RCX=RAX+RBX.
```

2. Множення:

```
а)  lea eax, [eax*2]          ; EAX = EAX x 2
    lea rax, [eax*2]          ; RAX = EAX x 2
    lea rax, [rax*2]          ; RAX = RAX x 2
б)  lea eax, [eax+eax*2]      ; EAX = EAX x 3
    lea rax, [eax+eax*2]      ; RAX = EAX x 3
    lea rax, [rax+rax*2]      ; RAX = RAX x 3
в)  lea rbx, [eax*4]          ; RBX = EAX x 4
г)  lea rcx, [ebx+ebx*4]      ; RCX = EBX x 5
д)  lea rax, [eax*8]          ; RAX = EAX x 8
е)  lea rcx, [ebx+ebx*8]      ; RCX = EBX x 9
ж) mov ebx, eax               ; EAX x 6
```

```
lea rax, [eax+eax*4]
lea rax, [eax+ebx]
```

з)
```
mov ebx, eax          ; EAX x 7
lea eax, [eax+eax*4]
lea eax, [eax+ebx*2]
```

и)
```
mov ebx, eax          ; EAX x 10
lea eax, [eax+eax*8]
lea eax, [eax+ebx]
```

3. Комбіновані багатоопераднi операції додавання і множення:

```
lea ecx, [eax+ebx*2+1]
```

2.5.1.2 Арифметичні команди

За допомогою асемблера можна розв'язувати будь-які завдання – *те, що неможливо зробити за допомогою асемблера, доводиться паяти, або неможливо зробити взагалі!* Однак, зазвичай, арифметичні команди використовуються як допоміжні для розв'язання абсолютно необчислювальних задач. Команди двійкової арифметики виконують операції *з числами з фіксованою комою*.

Суто обчислювальні задачі зазвичай пов'язані з виконанням операції з числами з плаваючою комою, і для цього у процесорах Intel64/AMD64 використовується спеціальний пристрій – Floating Point Unit (FPU), який розглянемо пізніше. У групі арифметичних команд виокремлюють три підгрупи команд.

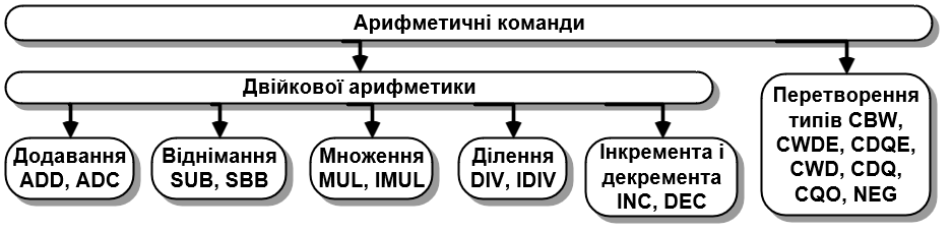


Рисунок 2.11 – Група арифметичних команд

Мнемоніка ADD|SUB <приймач>, <джерело>

Призначення	Арифметичне додавання|віднімання двійкових чисел
Алгоритм	<приймач> ← <приймач> + <джерело> | <приймач> ← <приймач> – <джерело>
Операнд	r – imm, m – imm, r – r, r – m, m – r
RFLAGS	OF, SF, ZF, AF, CF, PF

Мнемоніка ADC|SBB <приймач>, <джерело>

Призначення	Арифметичне додавання|віднімання окремих частин довгих двійкових чисел з урахуванням поміжрозрядних переносів|займів
Алгоритм	<приймач> ← <приймач> + <джерело> + CF | <приймач> ← <приймач> – (<джерело> + CF)
Операнд	r – imm, m – imm, r – r, r – m, m – r
RFLAGS	OF, SF, ZF, AF, CF, PF

Приклад. Додати до 64-розрядного числа в парі регістрів EDX:EAX (EDX – старші 32 розряди) число 0F3D672A564CD89A2h, а потім відняти його з результату.

```
mov ecx, 0F3D672A5h
mov ebx, 64CD89A2h
add eax, ebx
adc edx, ecx
sub eax, 64CD89A2h
sbb edx,0F3D672A5h
```

Арифметичне додавання|віднімання до|з двійкового числа одиниці – дуже поширена операція у програмуванні, і для її виконання в системі команд процесора передбачені спеціальні команди інкремента|декремента.

Мнемоніка INC|DEC <операнд>

Призначення	Інкремент|декремент <операнд>
Алгоритм	<операнд> ← <операнд> + 1| <операнд> ← <операнд> – 1
Операнд	r8/r16/r32/r64/m8/m16/m32/m64
RFLAGS	OF, SF, ZF, AF, PF відповідно до результату, *на CF – не впливають*

Мнемоніка MUL|IMUL <операнд>

Призначення	Арифметичне множення без урахування|з урахуванням знакових розрядів
Операнд	r8/r16/r32/r64, m8/m16/m32/m64
Алгоритм	<операнд> | Множене | Результат Байт | AL | AX←AL*операнд Слово | AX | DX:AX←AX *операнд Подвійне слово | EAX | EDX:EAX←EAX *операнд Зчетверене слово | RAX | RDX:RAX← RAX *операнд
RFLAGS	SF, ZF, AF, PF. Якщо старша половина результату нульова, то OF=CF=0, якщо не нульова (розширена знаковим бітом) – OF=CF=1.

Для команди IMUL існують двох і триоперандні формати, характерною ознакою яких є те, що вони не враховують подвоєння довжини результату і відсікають його старшу половину, якщо вона з’являється.

У двоопераднних командах IMUL r, r|m результат множення операндів розміщується на місце лівого операнда, який повинен бути 16/32/64-розрядним регістром. У триоперандних командах IMUL r, r|m, imm результат множення

правого і середнього операндів розміщується на місце лівого операнда, який повинен бути 16/32/64-розрядним регістром.

Мнемоніка DIV|IDIV <операнд>

<table>
<tr><td>Призначення</td><td>Арифметичне ділення без урахування|з урахуванням знакових розрядів</td></tr>
<tr><td>Операнд</td><td>r8/r16/r32/r64, m8/m16/m32/m64</td></tr>
<tr><td>Алгоритм</td><td>
<table>
<tr><td><операнд></td><td>Ділене</td><td>Частка</td><td>Залишок</td></tr>
<tr><td>Байт</td><td>AX</td><td>AL</td><td>AH</td></tr>
<tr><td>Слово</td><td>DX:AX</td><td>AX</td><td>DX</td></tr>
<tr><td>Подвійне слово</td><td>EDX:EAX</td><td>EAX</td><td>EDX</td></tr>
<tr><td>Зчетверене слово</td><td>RDX:RAX</td><td>RAX</td><td>RDX</td></tr>
</table>
</td></tr>
<tr><td>RFLAGS</td><td>SF, OF, SF, ZF, AF, PF не визначені.</td></tr>
</table>

Операнди розглянутих вище команд ADD/ADC/SUB/SBB повинні мати однаковий розмір. Але як діяти, якщо, наприклад, до слова потрібно додати байт? Очевидно, операнди необхідно попередньо вирівняти на однаковий розмір. Природно, що вирівнювання відкиданням «зайвих» бітів довшого операнда неприйнятно, тому що це змінить його значення. Отже, вирівнювати потрібно менший за розміром операнд убік більшого за розміром операнда.

На перший погляд, для цього досить записати у старших бітах числа, що вирівнюється, незначущі нульові біти: 10100110b → 00000000 10100110b. І це дійсно буде правильно, але тільки в тому випадку, *якщо розглядати число 10100110b як беззнакове*. Якщо ж уважати, що 10100110b – *це число зі знаком*, то «1» у старшому біті говорить про те, що воно негативне, і наявне в додатковому коді. Тоді розширення нулями призведе до того, що число з негативного перетворитися на позитивне, адже тепер у нього у старшому біті буде «0».

Розширення виду 10100110b → 10000000 10100110b теж призводить до невірного результату, у чому не важко переконатися, виконавши перетворення 10000000 10100110b з додаткового коду на прямий:

Вихідне : $(10100110b)_{\text{додатковий}} = (11011001b + 1)_{\text{прямий}} = (11011010b)_{\text{прямий}} = - (1\cdot 2^6 + 1\cdot 2^4 + 1\cdot 2^3 + 1\cdot 2^1) = - (64+16+8+2) = - 90d$

Розширене : $(10000000\ 10100110b)_{\text{додатковий}}$

$$\begin{array}{r} 11111111\ 01011001b \\ \hline 1 \\ \hline (11111111\ 01011010b)_{прямий} \end{array}$$

$(11111111\ 01011010b)_{\text{прямий}} = - (1\cdot 2^{14} + 1\cdot 2^{13} + 1\cdot 2^{12} + 1\cdot 2^{11} + 1\cdot 2^{10} + 1\cdot 2^9 + 1\cdot 2^8 + 1\cdot 2^6 + 1\cdot 2^4 + 1\cdot 2^3 + 1\cdot 2^1) = - 32\ 602d$

Результат неправильний через «зайві» одиниці у старших бітах числа. У випадку знакового числа правильним буде розширення виду 10100110b → 11111111 1010 0110b, тобто *розширення знаковим бітом* вихідного числа.

Виконавши перетворення 11111111 10100110b з додаткового коду на прямий, одержимо

$$\begin{array}{r} 11111111\ 10100110b \\ 10000000\ 01011001b \\ \hline 1 \\ \hline (10000000\ 01011010b)_{прямий} \end{array}$$

$(10000000\ 01011010b)_{\text{прямий}} = - (1\cdot 2^6 + 1\cdot 2^4 + 1\cdot 2^3 + 1\cdot 2^1) = - 90d$

Виконувати вирівнювання операндів може бути потрібно не тільки під час їхнього додавання й вирахування, але й під час множення й ділення, а загалом для тих або інших цілей, і для цього в системі команд передбачена група команд.

Мнемоніка CBW|CWDE|CDQE|CWD|CDQ|CQO

Призначення	Розширення *знаковим бітом* числа в регістрі-акумуляторі			
Алгоритм	Команда	розширює	до	
	CBW	AL	AX	

	CWDE	AX	EAX	
	CDQE	EAX	RAX	
	CWD	AX	DX:AX	
	CDQ	EAX	EDX:EAX	
	CQO	RAX	RDX:RAX	
Операнд	Визначаються кодом команди			
RFLAGS	Не впливають			

Після розгляду команд розширення, наразі розглянемо команди MOVSX і MOVZX, що були пропущені в групі команд пересилання. Ці команди дозволяють поєднати пересилання операнда з його розширенням, і, на відміну від CBW/CWDE/CDQE, місце розташування операнда яких фіксовано, дозволяють виконувати розширення в регістрі, відмінному від AL/AX/EAX/RAX.

Мнемоніка MOVSX|MOVSXD <приймач>, <джерело>

Мнемоніка	MOVSX <приймач>, <джерело>
Алгоритм	Пересилання в приймач розширеного знаком джерела
Операнд	r16–r8|m8, r32–r8|m8, r64–r8|m8, r32–r16|m16, r64–r16|m16
RFLAGS	Не впливає

Мнемоніка MOVZX <приймач>, <джерело>

Алгоритм	Пересилання в приймач розширеного нулем джерела
Операнд	r16–r8|m8, r32–r8|m8, r64–r8|m8, r32–r16|m16, r64–r16|m16
RFLAGS	Не впливає

Приклад 2.1. Для будь-яких *позитивних* значень змінних заданого типу: x, y, z – байт, u та v – слово, w – подвійне слово обчислити значення виразу $\frac{x^2+y^2+z^2+1}{u+v}+\frac{x^2\cdot y^2}{w-2\cdot v-u^2}$ і зберегти результат у змінній пам'яті result.

Доберемо тестовий набір даних, який ***для арифметики цілих чисел*** повинен надати абсолютний результат. Для x=2, y=3, z=4, u=5, v=5, w=71 значення виразу

$$\frac{2^2+3^2+4^2+1}{5+5}+\frac{2^2\cdot 3^2}{71-2\cdot 5-5^2}=\frac{4+9+16+1}{10}+\frac{4\cdot 9}{71-10-25}=\frac{30}{10}+\frac{36}{36}=3+1=4.$$

Наведений нижче аналіз виразу показує, що загалом за довільних позитивних значеннь чисел заданого розміру й *без урахування можливого переповнення унаслідок додавання* результат буде подвійним словом.

$$\frac{db^2+db^2+db^2+1}{dw+dw}+\frac{db^2\cdot db^2}{dd-2\cdot dw-dw^2}=\frac{dw+dw+dw+1}{dw}+\frac{dw\cdot dw}{dd-dd-dd}=\frac{dw}{dw}+\frac{dd}{dd}=$$

$$=\frac{dd}{dw}+\frac{dq}{dd}=dw+dd=dd$$

Вихідний код для розв'язання завдання:

```
.data
x       db 2
y       db 3
z       db 4
u       dw 5
v       dw 5
w       dd 71
.data?
result  dd ?
.code
main proc
; отримуємо по черзі x та y в квадраті та зберігаємо
```

```
; ці значення в регістрах для того щоб не обчислювати їх іще раз
; під час розрахунку чисельника другого дробу
        mov al,x        ; AL=x
        mul al          ; AX=AL*AL = x^2
        mov bx,ax       ; й зберегли квадрат x в BX
        mov al,y        ; AL = y
        mul al          ; AX = AL*AL = y^2
        mov cx,ax       ; й зберегли квадрат x в CX
        mov al,z        ; AL = z
        mul al          ; AX = AL*AL = z^2
        add ax,bx       ; AX = z^2+x^2
        add ax,cx       ; AX = z^2+x^2+y^2
        inc ax          ; AX = z^2+x^2+y^2+1
; Розширимо для подальшого ділення на слово
        cwd             ; DX:AX = z^2+x^2+y^2+1
; Обчислюємо знаменник
        movzx esi,u     ; ESI = u
        movzx edi,v     ; EDI = v
; SI = u та DI = v іще знадобляться  для подальших розрахунків
; Робимо робочу копію SI = u
        mov r8w,si      ; R8W = SI = u
        add r8w,di      ; R8W = u + v
; [...] – це позначення «ціла частина числа»
        div r8w         ; AX  = (DX:AX)/R8W = [(x^2+y^2+z^2+1)/(u+v)]
; Розширимо для подальшого додавання до подвійного слова
        cwde            ; EAX = [(x^2+y^2+z^2+1)/(u+v)]
 ; Звільнимо EAX який братиме участь у наступних розрахунках
        mov r8d,eax     ; R8D= [(x^2+y^2+z^2+1)/(u+v)]
        mov ax,bx       ; квадрат x зберігали в BX
```

```
        mul cx              ; квадрат у зберігали в CX. Отже DX:AX = x^2*y^2
        xchg dh,dl          ; збираємо результат  з пари DX:AX
        bswap edx           ; у єдине 32-розрядне в EDX
        mov dx,ax           ; для подальшого ділення
        mov ebx,edx         ; звільняємо EDX, який буде задіяний в подальшому
        mov ax, si          ; обчислюємо квадрат u (SI = u)
        mul ax              ; DX:AX = u^2
        xchg dh,dl          ; і збираємо результат
        bswap edx           ; із пари DX:AX
        mov dx,ax           ; у єдине 32-розрядне
        mov ecx,w           ; ECX = w
        sub ecx,edx         ; ECX = w - 2v
        lea edx,[edi+edi]   ; обчислюємо 2v (EDI = v)
        sub ecx,edx         ; ECX = w - 2v - u^2
        mov eax,ebx         ; EAX = x^2*y^2
        cdq                 ; розширюємо до EDX:EAX для ділення
        div ecx             ; EAX  = (EDX:EAX)/ECX = (x^2*y^2)/( w - 2v - u^2)
        add r8d,eax         ; додаємо EAX до першого дробу в R8D
        mov result, r8d     ; зберігаємо результат обчислень
    ret
main endp
end
```

2.5.1.3 Логічні команди

Виконання логічних команд, однойменних елементарним логічним функціям алгебри логіки NOT (інверсія), AND (кон'юнкція), OR (диз'юнкція), XOR (виключна диз'юнкція або сума за модулем два), зводиться до виконання *попарних побітових* операцій з відповідними бітами операндів, відповідно до таблиці істинності логічних функцій.

Біти		Функція		
b_i^1	b_i^2	b_i^1 OR b_i^2	b_i^1 AND b_i^2	b_i^1 XOR b_i^2
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	0

b_i^1 й b_i^2 - і-ті біти 1-го й 2-го операнда

Приклад

A= 01101011b A= 01101011b A= 01101011b

B= 11001001b B= 11001001b B= 11001001b

A OR B = 11101011b A AND B = 01001001b A XOR B = 10100010b

Мнемоніка NOT <операнд>

Алгоритм	Замінити усі нульові біти <операнд> на одиничні й навпаки
Операнд	r8/r16/r32/r64/m8/m16/m32/m64
RFLAGS	Не впливає

Мнемоніка AND <операнд> , <маска>

Призначення	Скидання окремих бітів або груп бітів операнда згідно з маскою
Алгоритм	<операнд> ← <операнд> & <маска>
Операнди	r-im, m-im, r-r, r-m, m-r
RFLAGS	OF=CF=0, а SF, ZF, PF відповідно до результату операції. AF не визначений

Мнемоніка TEST <операнд> , <маска>

Призначення	Перевірка значень певних бітів або груп бітів операнда
Алгоритм	Побітова кон'юнкція <операнд> і <маска> без збереження результату операції (тільки встановлення прапорців RFLAGS)
Операнди	r-im, m-im, r-r, r-m, m-r

RFLAGS	OF=CF=0, а SF, ZF, PF відповідно до результату операції. AF не визначений

Мнемоніка OR <операнд> , <маска>

Призначення	Установлення бітів або груп бітів операнда згідно з маскою
Алгоритм	Побітова диз'юнкція <операнд> та <маска>
Операнд	r-im, m-im, r-r, r-m, m-r
RFLAGS	OF=CF=0, а SF, ZF, PF відповідно до результату операції. AF не визначений

Мнемоніка XOR <операнд> , <маска>

Призначення	Перевірка, установлення, інверсія бітів або груп бітів операнда згідно з маскою
Алгоритм	Побітова сума за модулем 2 <операнд> та <маска>
Операнд	r-im, m-im, r-r, r-m, m-r
RFLAGS	OF=CF=0, а SF, ZF, PF відповідно до результату операції. AF не визначений

2.5.1.4 Команди зрушення

Команди зрушення поділяють на кілька підгруп.

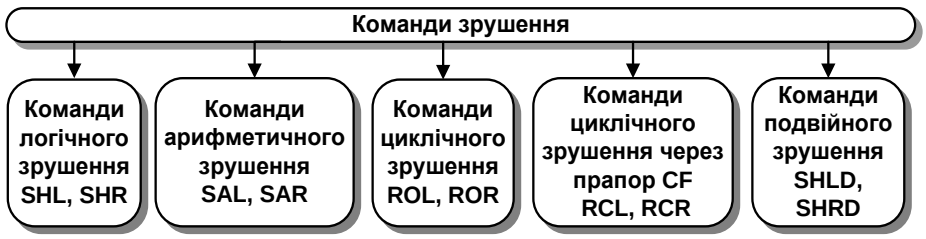


Рисунок 2.12 – Команди зрушення

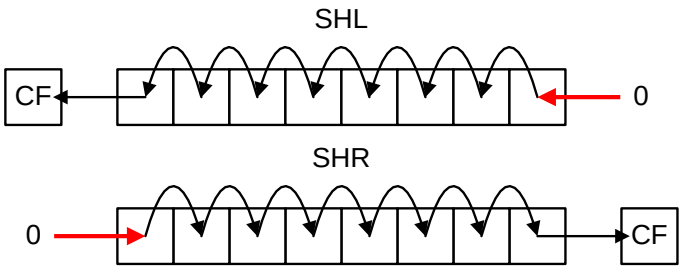
Ознакою, за якою виконують розподіл на підгрупи є те значення, яке використовується у якості біта і заповнює «вакантне» місце. *Для всіх команд*

зрушення прапор CF завжди встановлюється рівним біту, що останнім вийшов за межі операнда.

2.5.1.4.1 Команди логічного зрушення

В командах логічного зрушення бітом-заповнювачем є 0.

Мнемоніка SHL|SHR <операнд> , <кількість>

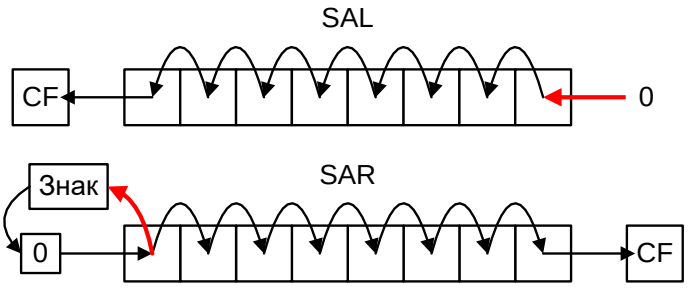
Призначення	Логічне зрушення операнда вліво|вправо на вказану <кількість> бітів
Алгоритм	
Операнд	r-im, m-im, r-CL, m-CL. В операнді, що визначає <кількість> зрушень у*раховуються лише 6 молодших бітів*
RFLAGS	Прапор CF завжди рівний біту, що останнім вийшов за межі операнда. Зрушення на 1 змінює значення прапора OF: SHL установлює його в 1, якщо після зсуву старший біт змінився (тобто старші два біти початкового числа не були однаковими), і в 0, якщо старший біт залишився тпким самим. SHR установлює OF у значення старшого біта початкового числа. Для зсувів на декілька бітів значення OF не визначено. Прапори SF, ZF, PF установлюються відповідно до результату, AF не визначений.

2.5.1.4.2 Команди арифметичного зрушення

Логічне зрушення вліво/вправо на 1 двійковий розряд *беззнакового числа* еквівалентно збільшенню/зменшенню числа в 2 рази. Логічне зрушення *вправо знакового числа* може призвести до зміни його знакового розряду – негативне

число може перетворитися на позитивне. Для виконання швидкого множення/ділення на 2^k в систему команд додано команди арифметичного зрушення.

Мнемоніка SAL|SAR <операнд> , <кількість>

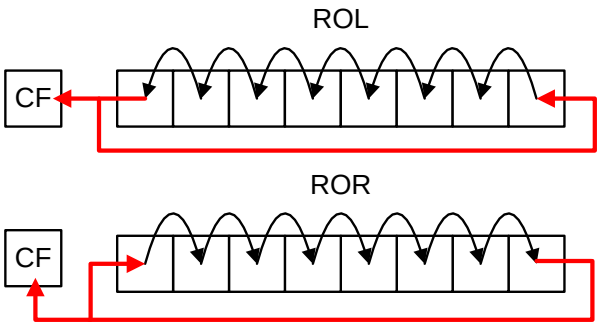
Призначення	Арифметичне зрушення <операнд> вліво|вправо на вказану <кількість> бітів
Алгоритм	
Операнд	r-im, m-im, r-CL, m-CL. В <операнд>, що визначає кількість зрушень, ураховуються *лише 6 молодших бітів* (5 у 32-бітному режимі)
RFLAGS	Аналогічно команді SHL|SHR, за винятком того, що внаслідок зсуву на 1 біт значення прапора OF завжди встановлюється у нульове

2.5.1.4.3 Команди циклічного зрушення

На відміну від команд логічного та арифметичного зрушення, команди циклічного зрушення не призводять до втрат бітів операнда – вони тільки змінюють їх порядок.

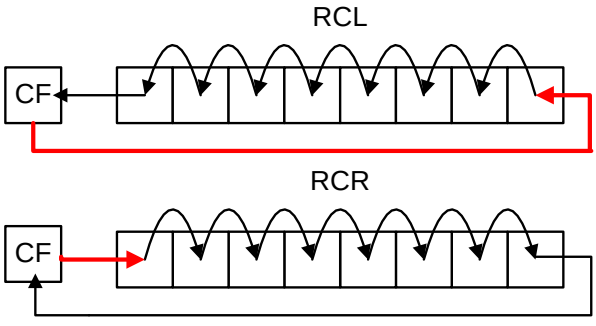
Мнемоніка ROL|ROR <операнд> , <кількість>

Призначення	Циклічне зрушення операнда вліво|вправо на вказану кількість бітів

Алгоритм	
Операнд	r-im, m-im, r-CL, m-CL. В операнді, що визначає кількість зрушень ураховуються лише 6 молодших бітів (5 в 32-бітному режимі)
RFLAGS	Прапор CF завжди рівний біту, що останнім вийшов за межі операнда, прапор OF визначений тільки для зсувів на 1: він установлюється, якщо змінилося значення найстаршого біта, і скидається, якщо старший біт не змінився. Прапори SF, ZF, AF і PF не змінюються.

2.5.1.4.4 Команди циклічного зрушення через прапор CF

Мнемоніка RCL|RCR <операнд> , <кількість>

Призначення	Циклічне зрушення <операнд> вліво|вправо через прапор CF на вказану <кількість> бітів
Алгоритм	

Операнд	r-im, m-im, r-CL, m-CL. В операнді, що визначає кількість зрушень, ураховуються лише 6 молодших бітів
RFLAGS	Аналогічно командам ROL|ROR

2.5.1.4.5 Команди подвійного зрушення

Команди подвійного зрушення у комбінації з іншими командами та команди RCL|RCR дозволяють виконувати операції з числами, що перевищують апаратно підтримувані процесором типи даних.

Мнемоніка	SHLD|SHRD <приймач> , <джерело> , <кількість>
Призначення	Логічне зрушення подвійного слова у <приймач> вліво|вправо на вказану <кількість> бітів
Алгоритм	1. Зрушити подвійне слово у <приймач> логічним зрушенням вліво|вправо на один біт, одночасно встановлюючи значення молодшого|старшого біта <приймач> значенням старшого|молодшого біта <джерело> та зрушуючи вліво|вправо на 1 біт <джерело>. 2. Установити прапор CF у попереднє значення старшого|молодшого біта <приймач>. 3. Повторити попередні 2 дії визначену <кількість> разів. 4. *Поновити початковий стан джерела*.
Операнд	r - m16, r16, imm8; r - m16, r16, CL; r - m32, r32, imm8; r - m64, r64, imm8; r - m32, r32, CL; r - m64, r64, CL
RFLAGS	Прапори SF, ZF, PF установлюються усіма зсувами відповідно до результату в приймачі, значення AF та OF не визначено

Основне призначення логічних команд полягає у виконанні операцій з окремими бітами або групами бітів у межах операнда:

1. Для встановлення в 1, або скидання в 0 заданих бітів:
   а) and eax,7FFFFFFFh ; скинути старший біт EAX
   б) or eax,80000001h ; установити старший і молодший біти EAX

2. Для встановлення певного значення в групах заданих бітів:

а) and eax,0FFFFFF0Fh ; скинути другу тетраду EAX
or eax,20h ; і занести в неї 0010 0000b

б) and eax,0FFFFF47Fh ; скинути 7,8 і 9-й біти (0100 0111 1111)
or eax, 080h ; і занести в них 1000 0000b

3. Для перевірки значень окремих бітів або груп бітів

а) test eax,80000000h ; старший біт EAX = 1?

б) test rax,3 ; 2 молодші біти (0011) RAX одиничні ?

в) test rax,1 ; у RAX парне число ?

4. Для отримання нульового значення регістра краще використовувати команду XOR:

а) mov rax, 0 ; має довгий код, не впливає на RFLAGS

б) sub rax, rax ; повільна й змінює RFLAGS

в) and rax, 0 ; швидка, має довгий код і змінює RFLAGS

г) xor rax, rax ; швидка і коротка, змінює RFLAGS

5. Для розширення *беззнакового* числа в будь-якому регістрі, наприклад, BL/BX/EBX можна скористатися логічними командами замість команд CBW/CWD/CWDE/CDQ місце положення операнда яких фіксоване:

xor bh, bh ; беззнакове розширення BL до BX

and ebx, 000000FFh ; беззнакове розширення BL до EBX

and ebx, 0000FFFFh ; беззнакове розширення BX до EBX

xor dx, dx ; без знакове розширення AX до DX:AX

6. Для локалізації окремих груп бітів для подальшої обробки використовуються логічні команди в комбінації із командами зрушення:

а) and rax,7 ; тепер у RAX/EAX/AX/AL залишилися 3 молодші біти

б) shr eax,16 ; тепер у EAX/AX старші 16 бітів EAX

в) shr rax,32 ; тепер у RAX/EAX старші 32 біта RAX

г) and eax,0F0h ; виділити значення старшої тетради EAX
shr al, 4 ; і перемістити в 4 молодші біти EAX/AX/AL

7. Для конкатенації 16-розрядних регістрів у єдиний 32-розрядний регістр:

а) результат в EAX

```
mul bx
shl eax, 16
or ax, dx
ror eax,16
```

б) результат в EDX

```
mul bx
shl edx, 16
or dx, ax
```

8. Для конкатенації 32-розрядних регістрів у єдиний 64-розрядний регістр:

а) результат в RAX

```
mul ebx
shl rax, 32
or eax, edx
ror rax,32
```

б) результат у RDX

```
mul ebx
shl rdx, 32
or edx, eax
```

9. Для обміну значень старшої та молодшої половини регістра можна використовувати команди ROR/ROL:

а) ror ax, 8 ; краще xchg al, ah

б) ror eax,16 ; xchg тут неможливо

10. Для виконання швидкого множення/ділення на 2^k краще використовувати команди SHL/SHR/SAR:

а) shl eax, 1 ; eax*2 для числа без знаку

б) shr ebx, 1 ; ebx/2 для числа без знаку

в) sar ebx, 1 ; ebx/2 для числа зі знаком

11. Для виконання операції з довгими числами

```
sar edx, 1      ; edx:eax/2
rcr eax, 1
```

2.5.1.5 Команди передавання керування

Передача керування в програмі полягає у зміні значення регістра RIP, що забезпечує вибірку на виконання не наступної команди, що слідує за поточною виконуваною, а деякої іншої. До апаратних засобів, які забезпечують передавання керування у програмах належить регістр RIP і певним чином регістр RSP. Програмна реалізація передавання керування у програмі може бути виконана за допомогою команд однієї із трьох груп.

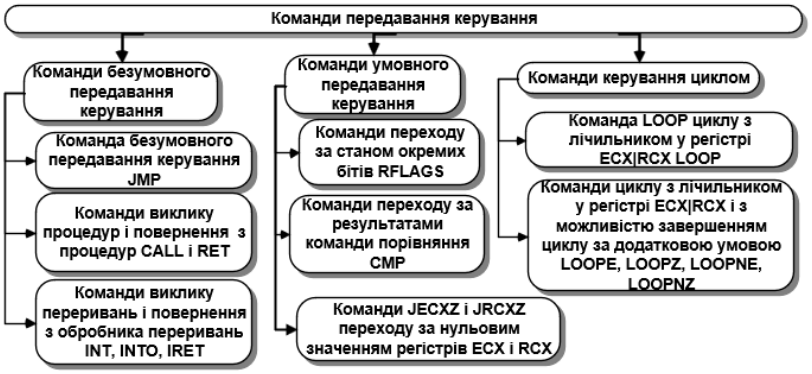
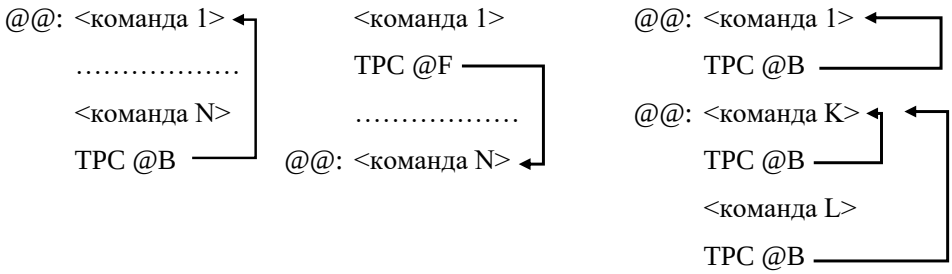


Рисунок 2.13 – Команди передавання керування

Як операнди команд передавання керування, тобто для позначення в програмі місця, в яке буде виконано передавання керування, *найчастіше* (але не обов'язково) використовуються мітки. Мітку можна визначити оператором ":" (двокрапка). Синтаксис:

```
<ім'я>:     <команда 1>
            ……
            <команда N>
```

Асемблер MASM припускає використання локальних (безіменних) міток "@@:", кількість яких може бути довільна. Для посилання на такі мітки в командах використовуються позначення @F/@B, що позначають уперед (@F – forward)/назад (@B – back) на ***найближчу*** (безіменну) мітку @@: відносно команди, у якій вони є.



«TPC» позначає узагальнене мнемонічне позначення ***команд передавання керування***.

2.5.1.5.1 Команди безумовного передавання керування

Із цієї групи команд ми розглянемо тільки команди JMP, CALL і RET.

Мнемоніка JMP операнд

Призначення	Перехід на виконання команди, адреса якої визначається операндом
Алгоритм	Завантажує в RIP значення, що визначається її операндом
Операнд	rel8, rel32, r64, m64, m16:16, m16:32, m16:64
RFLAGS	Не впливає (окрім випадку перемикання задач)

Приклад. Усі наведені нижче команди JMP виконують передавання керування в одне і те саме місце, демонструючи різні форми операнда команди

```
.data?
ma          dq ?
.code
main proc
```

```
        jmp m
        jmp short m         ; rel8
        lea rax,m
        jmp rax
        mov ma,rax
        jmp [ma]
        lea rax,ma
        jmp qword ptr [rax]
        jmp $+5+256         ; машинний код цієї команди ; E9 00010000 (5 байтів)
        db 256 dup(90h)     ; це 00000100h байтів машинного коду команди NOP
m:      ret
        main endp
end
```

Мнемоніка CALL операнд

Призначення	Виклик підпрограми
Алгоритм	Зберігає на верхівці стека подібно команді PUSH адресу наступної за нею команди і змінює значення RIP на те, що визначається її операндом
Операнд	rel32, r64, m64, m16:16, m16:32, m16:64
RFLAGS	не впливає окрім випадку перемикання задач

Мнемоніка RET [<число>]

Призначення	Повернення з підпрограми
Алгоритм	Зчитує значення з верхівки стека в RIP подібно команді POP. Якщо операнд <число> присутній, то команда додатково виконує переміщення верхівки стека RSP = RSP + <число>
Операнд	imm16

RFLAGS	Не впливає, окрім випадку перемикання задач

Приклад. Усі наведені нижче команди CALL виконують передавання керування в одне і те саме місце, демонструючи різні форми операнда команди.

```
.data?
ProcAddr    dq ?
.code
        main proc
        call ProcName
        lea rbx,ProcName
        call rbx
        mov ProcAddr,rbx
        call [ProcAddr]
        lea rbx,ProcAddr
        call qword ptr [rbx]
        lea rbx, retaddr
        push rbx
        jmp ProcName
        retaddr:
        ret
ProcName:
        ret
        main endp
end
```

Практичне використання команд CALL/RET розглянемо пізніше, у межах теми, про процедурне програмування.

2.5.1.5.2 Команди умовного передавання керування

Основною апаратною складовою, що забезпечує практичну реалізацію умовного передавання керування, є регістр прапорців мікропроцесора. *Значення прапорців регістра RFLAGS у кожен момент часу відображають стан самого мікропроцесора, і програми, що він виконує*. Мікропроцесор «повідомляє» про результат виконання кожної команди установкою тих або інших прапорців.

Під час опису команд завжди відзначаються ті прапорці, які можуть бути змінені внаслідок виконання команди, адже це надає *можливість аналізувати стан програми, і втручатися в порядок її виконання за допомогою команд умовного передавання керування*.

Група команд умовного передавання керування налічує декілька різних мнемонічних позначень команд поєднаних загальним алгоритмом і спільним набором операндів. Для опису команд цієї групи використовується узагальнене мнемонічне позначення Jcc, утворене додаванням до букви «J» (від Jump – стрибнути) мнемонічного позначення умови «cc» унаслідок виконання якої цей «стрибок» виконується.

Мнемоніка Jcc операнд

Призначення	Аналіз поточного стану виконання коду додатка з метою можливої зміни подальшого шляху виконання
Алгоритм	1. Перевірити умову, що визначається кодом операції (мнемонічним позначенням умови “cc”) 2. Якщо умова виконується, то передати керування на адресу, що визначається операндом команди Jcc 3. Якщо умова не виконується, то передати керування наступній за Jcc команді
Операнд	rel8, rel32
RFLAGS	Не впливає (можливе використання декількох команд Jcc одна за іншою)

Умовою переходу завжди є певний стан тих або інших прапорців регістра RFLAGS. Деякі з команд приймають рішення щодо виконання переходу, перевіряючи стан одного конкретного біта в RFLAGS, а інші виконують складні перевірки *певного одночасного стану декількох прапорців* RFLAGS. Розгляд групи команд умовного передавання керування почнемо з найпростіших.

2.5.1.5.2.1 Команди переходу за станом окремих бітів RFLAGS

Мнемоніки команд передавання керування за станом окремих бітів RFLAGS можна оримати додаванням до букви «J» першої букви назви прапорця RFLAGS, за одиничного значення якого здійснюється перехід. Якщо перехід необхідно виконати за нульового значення прапорця, то до букви «J» додається ще буква «N» (від Not - не). Рисунок 2.14 демонструє схему формування мнемоник команд переходу за станом окремих бітів RFLAGS.

$$\text{J (ump if) [N(ot)]}\begin{Bmatrix}\text{C (F)}\\ \text{P (F)}\\ \text{Z (F)}\\ \text{S (F)}\\ \text{O (F)}\end{Bmatrix}$$

Рисунок 2.14 – Схема формування мнемоник команд переходу

Таблиця 2.3 містить мнемонічні позначення команд і умови, за яких вони виконують передавання керування.

Таблиця 2.3

JC	JP	JZ	JS	JO	JNC	JNP	JNZ	JNS	JNO
CF=1	PF=1	ZF=1	SF=1	OF=1	CF=0	PF=0	ZF=0	SF=0	OF=0

2.5.1.5.2.2 Команди переходу за результатом команди порівняння

Рішення про передавання керування в програмі може бути прийнято на підставі операцій порівняння виду «==», «!=», «<», «<=», «>», «>=», які є звичайними для мов програмування високого рівня.

В основу алгоритму порівняння, який можна реалізувати апаратними засобами, покладено виконання операції віднімання. Якщо, наприклад, після виконання команди SUB EAX,EBX значення регістра EAX виявиться нульовим, про що мікропроцесор відразу ж повідомить установленням в 1 прапорця ZF в RFLAGS, то зовсім очевидно, що до операції значення в регістрі EAX дорівнювало значенню у регістрі EBX.

Команда SUB руйнує значення операнда-приймача, що не зовсім зручно для програмування, і тому в системі команд мікропроцесора для виконання порівняння відніманням передбачена спеціальна команда CMP. Ця команда, відрізняється від команди SUB тільки тим, що *нікуди не розміщує результат віднімання, а тільки встановлює прапорці*.

Аналізуючи стан прапорців регістра RFLAGS після віднімання різних за співвідношеннями операндів і *з урахуванням їх знаків*, можна дійти висновку, що для кожного випадку є характерний і властивий тільки йому певний стан тих або інших прапорців. За результатами проведеного аналізу до системи команд включений набір команд передавання керування за результатами виконання команди порівняння відніманням. Це команди передавання керування за станом регістра RFLAGS, власне, комплексної дії. Рішення про передавання керування приймається відповідно до стану декількох прапорців, характерних для результату виконання команди CMP за певного співвідношення значень операндів і *з урахуванням їх знаків*.

Таблиця 2.4 відображає стан прапорців RFLAGS для передавання керування після виконання команди CMP.

Таблиця 2.4

Тип операндів у команді CMP операнд1, операнд2	Команда	Виконує перехід якщо	Стан прапорців для виконання переходу
Зі знаком	JL/JNGE	операнд1< операнд2	SF<>OF
Зі знаком	JLE/JNG	операнд1<=операнд2	SF<>OF або ZF=1
Зі знаком	JG/JNLE	операнд1>операнд2	SF=OF і ZF=0
Зі знаком	JGE/JNL	операнд1>=операнд2	SF=OF
Без знаку	JB/JNAE	операнд1<операнд2	CF=1
Без знаку	JBE/JNA	операнд1<=операнд2	CF=1 або ZF=1
Без знаку	JA/JNBE	операнд1 > операнд2	CF=ZF=0
Без знаку	JAE/JNB	операнд1>=операнд2	CF=0
Будь-які	JE	операнд1 = операнд2	ZF=1
Будь-які	JNE	операнд1<>операнд2	ZF=0

Рисунок 2.156 демонструє схему формування мнемоник команд переходу за станом бітів RFLAGS після виконання команди CMP.

$$\text{J (ump if) [N(ot)]}\begin{Bmatrix}\text{A (bove)}\\ \text{B (elow)}\\ \text{G (reat)}\\ \text{L (ess)}\end{Bmatrix}\text{ [(or) E (qual)]}$$

Рисунок 2.156 – Схема формування мнемоник команд переходу за станом бітів RFLAGS

Варіанти мнемонік команд Jcc, які використовуються після команди CMP, легко запам’ятати за вищенаведеною схемою:

- Equal – дорівнює/не дорівнює (JE == JZ, JNE == JNZ);
- Below – нижче (менше) *для беззнакових* (JB == JC);

– Below or Equal – нижче (менше) або дорівнює *для без знакових* (JBE);

– Above or equal – вище (більше) або дорівнює *для беззнакових* (JAE == JNB == JNC);

– Less – менше **для знакових** (JL);

– Great – більше **для знакових** (JG);

– Not great or equal – не більше або дорівнює **для знакових** (JNGE == JL).

2.5.1.5.3 Команди керування циклом

В загальному випадку будь-який цикл можна організувати з урахуванням попередньо розглянутих команд умовного передавання керування, однак у системі команд процесора є і спеціально передбачені для цього команди.

Мнемоніка LOOP мітка

Призначення	Організація циклу з лічильником у регістрі RCX/ECX/CX
Алгоритм	Зменшити на 1 значення в регістрі RCX. Якщо RCX не дорівнює нулю, то перейти на команду, позначену міткою, інакше на наступну за LOOP команду.
Операнд	rel8 (зміщення до мітки повинно бути в діапазоні -128…+127 байтів від команди)
RFLAGS	Не впливає

Мнемоніка LOOPE|LOOPNE мітка

Призначення	Організація циклу з лічильником у регістрі RCX і додатковою умовою
Алгоритм	Зменшити на 1 значення RCX. Якщо RCX не дорівнює нулю і $ZF = \begin{cases} 1 \text{ для LOOPE} \\ 0 \text{ для LOOPNE} \end{cases}$, то перейти на команду, позначену міткою, інакше – на наступну за LOOPE|LOOPNE команду. Зміщення до мітки повинно бути в діапазоні -128…+127 байтів.

Операнд	rel8
RFLAGS	Не впливає

Зазвичай команда LOOP використовується за такою схемою:

```
                mov rcx, <кількість повторень циклу>
<мітка циклу>:  ; . . . . . .
                ; тіло циклу
                loop <мітка циклу>
```

LOOP може виконувати *тільки короткий (short) перехід*, тобто розмір машинного коду команд тіла циклу, розташованих поміж командою LOOP і міткою-операндом не повинен перевищувати 128 байт. Такого обмеження можна уникнути замінивши команду LOOP еквівалентною послідовністю:

```
                mov rcx, <кількість повторень циклу>
<мітка циклу>:  ; . . . . . .
                ; тіло циклу
                ; . . . . . .
                dec rcx
                jnz <мітка циклу>
```

2.5.1.6 Команди обробки ланцюжків

Ланцюжок являє собою розміщену в пам’яті неперервну послідовність елементів однакового типу – байтів, слів, подвійних або зчетверених слів.

Для мікропроцесора байдуже цільове призначення елементів ланцюжка, для нього вся пам’ять розглядається як ланцюжок байт.

У системі команд мікропроцесора існує п’ять базових операцій обробки ланцюжків (Рисунок 2.167). Кожна з них реалізується в мікропроцесорі чотирма командами, мнемоніки яких відображають розмір елемента ланцюжка – байт (B), слово (W), подвійне слово (D) або зчетверене слово (Q).

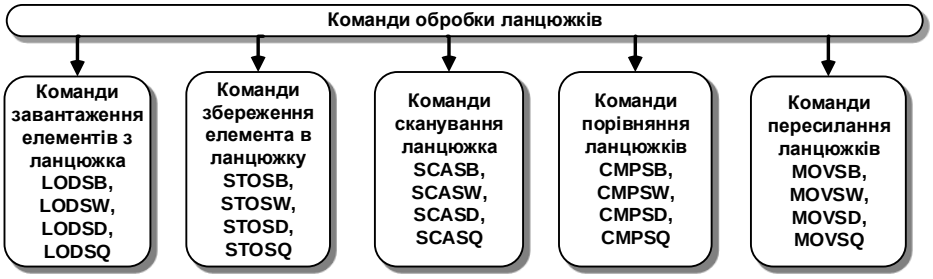


Рисунок 2.167 – Команди обробки ланцюжків

Для адресації в пам’яті елементів ланцюжків команди їх обробки використовують регістри RSI та RDI, оперуючи для цього такими поняттями, як *ланцюжок – джерело та ланцюжок – приймач*.

Джерело – це ланцюжок, адреси елементів якого команди обробки ланцюжків визначають за вмістом регістрів DS:(R|E)SI (SI – Source Index).

Приймач – це ланцюжок, адреси елементів якого команди обробки ланцюжків визначають за вмістом регістрів ES: (R|E)DI (DI – Destination Index).

2.5.1.6.1 Особливості адресації ланцюжків

Особливістю команд обробки ланцюжків є використання автоінкрементної або автодекрементної адресації. Алгоритм роботи *будь-якої команди обробки ланцюжків* обов’язково передбачає *модифікацію значення або регістра RSI, або регістра RDI*, або одночасно пари регістрів (R|E)SI та (R|E)DI так, щоб вони після виконання команди містили адресу наступного або наступних елементів ланцюжків.

Обробка елементів ланцюжка у разі автоінкрементної адресації відбувається у бік старших адрес пам’яті, тобто команди обробки ланцюжків збільшують значення або регістра (R|E)SI, або регістра (R|E)DI, або пари регістрів (R|E)SI та (R|E)DI одночасно.

Обробка елементів ланцюжка у разі автодекрементній адресації відбувається у бік молодших адрес пам’яті, тобто команди обробки ланцюжків

зменшують значення або регістра (R|E)SI, або регістра (R|E)DI, або пари регістрів (R|E)SI та (R|E)DI одночасно.

Напрямок обробки залежить від значення прапорця DF регістра прапорців RFLAGS:

$$DF = \begin{cases} 0, \text{ автоінкрементна адресація (обробка у бік старших адрес)} \\ 1, \text{ автодекрементна адресація (обробка у бік молодших адрес)} \end{cases}$$

Команда CLD установлює DF=0, що визначає напрямок обробки елементів ланцюжка з молодших адрес пам'яті у старші завдяки *автоматичному збільшенню* командами обробки ланцюжків значень індексних регістрів (R|E)SI та/або (R|E)DI на число, що дорівнює розміру в байтах елементів ланцюжків.

Команда STD установлює DF=1, що визначає напрямок обробки елементів ланцюжка зі старших адрес пам'яті в молодші завдяки *автоматичному зменшенню* значень індексних регістрів (R|E)SI та/або (R|E)DI на число, що дорівнює розміру в байтах елементів ланцюжків.

У 64-розрядній ОС Windows NT сегментні регістри DS та ES не використовується і завантаження селектора в сегментний регістр виконувати не потрібно. У подальшому в описі команд обробки ланцюжків сегментні регістри DS та ES не зазначаються.

2.5.1.6.2 Команди завантаження елементів з ланцюжка

Команда $\begin{cases} \text{LODSB} \\ \text{LODSW} \\ \text{LODSD} \\ \text{LODSQ} \end{cases}$ копіює $\begin{cases} \text{байт} \\ \text{слово} \\ \text{подвійне слово} \\ \text{зчетверене слово} \end{cases}$ із комірки пам'яті за адресою в регістрі ESI|RSI в регістр $\begin{cases} \text{AL} \\ \text{AX} \\ \text{EAX} \\ \text{RAX} \end{cases}$ і $\begin{cases} \text{збільшує, якщо DF=0} \\ \text{зменшує, якщо DF=1} \end{cases}$ значення

регістра ESI|RSI на $\begin{cases}1\\2\\4\\8\end{cases}$ не змінюючи значення регістра прапорців RFLAGS

2.5.1.6.3 Команди збереження елемента в ланцюжку

Команда $\begin{cases}\text{STOSB}\\\text{STOSW}\\\text{STOSD}\\\text{STOSQ}\end{cases}$ копіює $\begin{cases}\text{байт}\\\text{слово}\\\text{подвійне слово}\\\text{зчетверене слово}\end{cases}$ з регістру $\begin{cases}\text{AL}\\\text{AX}\\\text{EAX}\\\text{RAX}\end{cases}$ у комірку

пам’яті за адресою EDI|RDI і $\begin{cases}\text{збільшує, якщо DF=0}\\\text{зменшує, якщо DF=1}\end{cases}$ значення регістра EDI|RDI

на $\begin{cases}1\\2\\4\\8\end{cases}$ не змінюючи значення регістра прапорців RFLAGS

2.5.1.6.4 Команди сканування ланцюжка

Команда $\begin{cases}\text{SCASB}\\\text{SCASW}\\\text{SCASD}\\\text{SCASQ}\end{cases}$ вираховує з регістру $\begin{cases}\text{AL}\\\text{AX}\\\text{EAX}\\\text{RAX}\end{cases}$ значення $\begin{cases}\text{байта}\\\text{слова}\\\text{подвійного слова}\\\text{зчетвереного слова}\end{cases}$ в

пам’яті за адресою в регістрах EDI|RDI, $\begin{cases}\text{збільшує, якщо DF=0}\\\text{зменшує, якщо DF=1}\end{cases}$ значення

регістра EDI|RDI на $\begin{cases}1\\2\\4\\8\end{cases}$ і встановлює прапорці OF, SF, ZF, AF, PF, CF відповідно

до результату операції, *не зберігаючи сам результат*.

2.5.1.6.5 Команди порівняння ланцюжків

Команда $\begin{cases}\text{CMPSB}\\ \text{CMPSW}\\ \text{CMPSD}\\ \text{CMPSQ}\end{cases}$ вираховує значення $\begin{cases}\text{байта}\\ \text{слова}\\ \text{подвійного слова}\\ \text{зчетвереного слова}\end{cases}$ в комірці пам’яті за адресою в регістрах ESI|RSI і значення $\begin{cases}\text{байта}\\ \text{слова}\\ \text{подвійного слова}\\ \text{зчетвереного слова}\end{cases}$ в комірці пам’яті за адресою в регістрах EDI|RDI, $\begin{cases}\text{збільшує, якщо DF=0}\\ \text{зменшує, якщо DF=1}\end{cases}$ значення регістрів ESI|RSI та EDI|RDI на $\begin{cases}1\\ 2\\ 4\\ 8\end{cases}$ і встановлює прапорці відповідно до результату операції, *не зберігаючи сам результат*.

2.5.1.6.6 Команди пересилання ланцюжків

Команда $\begin{cases}\text{MOVSB}\\ \text{MOVSW}\\ \text{MOVSD}\\ \text{MOVSQ}\end{cases}$ копіює значення $\begin{cases}\text{байта}\\ \text{слова}\\ \text{подвійного слова}\\ \text{зчетвереного слова}\end{cases}$ із комірки пам’яті за адресою в регістрах (DS:ESI)|RSI в комірку пам’яті за адресою в EDI|RDI і $\begin{cases}\text{збільшує, якщо DF=0}\\ \text{зменшує, якщо DF=1}\end{cases}$ значення регістрів ESI|RSI та EDI|RDI на $\begin{cases}1\\ 2\\ 4\\ 8\end{cases}$

2.5.1.6.7 Префікси повторення

Автоматична зміна вмісту регістрів ESI|RSI та|або EDI|RDI абсолютно всіма командами обробки ланцюжків робить їх зручними для використання в циклах, для чого, власне, це й передбачалося. Наприклад, пошук у ланцюжку певного значення без використання і з використання команд обробки ланцюжків може виконуватися так:

```
        mov rcx,<довжина ланцюжка>      mov rcx,<довжина ланцюжка>
        lea rdi,<приймач>               lea rdi,<приймач>
        mov al,<значення для пошуку>    mov al,<значення для пошуку>
@@:     cmp al,byte ptr [rdi]           @@:  scasb
        jz @F                                loopne @B
        inc rdi                         @@:  …….
        loop @B
@@:  …….
```

Як видно з прикладу, використання автоінкрементної/автодекрементної адресації, властивої командам обробки ланцюжків, зменшує кількість використаних команд.

Іще більшому скороченню коду сприяє використання команд обробки ланцюжків, обидва операнди яких можуть розташовуватися у пам'яті. Наприклад, пересилання з одного ланцюжка байтів в інший без використання і з використанням команд обробки ланцюжків може виконуватися так:

```
        mov rcx,<довжина ланцюжка>      mov rcx,<довжина ланцюжка>
        lea rsi,<джерело>               lea rsi,<джерело>
        lea rdi,<приймач>               lea rdi,<приймач>
@@:     mov al, byte ptr [rsi]     @@:  movsb
        mov byte ptr [rdi],al           loop @B
        inc rsi
        inc rdi
        loop @B
```

Як видно з прикладів, при використанні команд обробки ланцюжків, тіло циклу складається з єдиної команди. Для розв'язання типових задач, що стосуються обробки ланцюжків, замість команд передавання керування, краще використовувати так звані префікси повторення.

Префікс REP примушує командну обробку ланцюжків повторюватися, поки RCX не дорівнюватиме нулю (RCX раз).

Префікс REPE|REPZ примушує командну обробки ланцюгів повторюватися, поки RCX не дорівнюватиме нулю і прапорець нуля встановлений (ZF=1).

Префікс REPNE|REPNZ примушує командну обробки ланцюгів повторюватися поки RCX не дорівнюватиме нулю і прапорець нуля скинуто (ZF=0).

Із використанням префіксів повторення попередні приклади матимуть такий вигляд:

```
; Пересилання ланцюжка                ; Пошук у ланцюжку
mov rcx,<довжина ланцюжка>            mov rcx,<довжина ланцюжка>
lea rsi,<джерело>                     lea rdi,<приймач>
lea rdi,<приймач>                     mov al,<значення для пошуку>
rep movsb                             repne scasb
```

3 ОБРОБКА СТРУКТУРОВАНИХ ТИПІВ ДАНИХ

3.1 Одновимірні масиви

Масив – це структурований тип даних, що являє собою множину з певною кількістю елементів *однакового типу*. Одновимірні масиви можна назвати природною для процесору формою подання даних, оскільки пам'ять має лінійну адресну структуру і її можна розглядати як одновимірний масив.

Для роботи з масивами зокрема, а загалом з будь-якими структурованими даними, зручно використовувати оператори MASM, які описані у таблиці 3.1.

Таблиця 3.1 – Оператори MASM для роботи з типами даних

Оператор	Опис
SIZE змінна	Повертає число байтів у змінній, що було виділено під час її початкової ініціалізації.
SIZEOF змінна|тип	Повертає число байтів, що займає змінна або тип.
LENGTH змінна	Повертає число елементів даних у змінній, що було створено під час її початкової ініціалізації.
LENGTHOF змінна	Повертає число об’єктів даних у змінній.
TYPE вираз	Повертає тип (розмір у байтах) виразу.

Обробка одновимірних масивів, зазвичай, передбачає використання принаймні одного циклу, а загалом і вкладених циклів. Для звернення до елементів масиву можна використовувати різні способи адресації операндів у пам’яті. Вибір конкретного способу адресації залежить від розв’язуваної задачі.

Приклад 3.1. Визначення суми позитивних елементів у масиві байт і кількості негативних елементів у масиві слів.

```
.data
x       db 1,2,3,-4,-5,6,0,7,8,0,5,8        ; масив байт
y       dw 1,2,3,-4,-5,6,0,7,8,0,5,0        ; масив слів
.code
main proc
        lea rsi,x       ;завантаження в RSI адреси початку (1-го елемента) масиву x
        xor rbx,rbx  ; у RBX буде накопичуватися сума
        mov rcx, lengthof x         ; у RCX кількість елементів масиву x
m1:     cmp byte ptr [rsi],0        ; порівняння поточного елемента з нулем
        jle m2                      ; пропускаємо негативний елемент
        add bl, byte ptr [rsi]      ; додаємо до суми з урахуванням
        adc bh,0                    ; можливого переповнення
```

```
m2:   inc rsi          ; перераховуємо адресу наступного елемента масиву
      loop m1          ; продовження циклу, якщо RCX іще ненульовий
;### після завершення циклу в RBX сума елементів масиву x ###
      xor rdi,rdi      ; початкове значення кількості негативних елементів у масиві
      lea rsi,y        ;завантаження в RSI адреси початку (1-го елемента) масиву y
      mov rcx,lengthof y       ; у RCX кількість елементів масиву y
m3:   cmp word ptr [rsi],0     ; порівняння поточного елемента з нулем
      jge m4                   ; пропускаємо позитивний елемент
      inc rdi                  ; збільшення лічильника
m4:   add rsi, type y    ; перераховуємо адресу наступного елемента масиву
      loop m3            ; продовження циклу, якщо RCX ще не нульовий
      ret
main endp
end
```

Приклад 3.1. Визначення суми позитивних елементів з парними індексами одновимірного масиву слів.

Загалом, за довільного місцерозташування масиву в пам'яті адреси його елементів можуть бути як парними, так і непарними числами, що збільшуватимуться для масиву слів із кроком 2. Отже, парність чи не парність індексу елемента масиву не можна визначати, перевіривши парність чи непарність його адреси.

Для розв'язання цієї задачі можна скористатися базовою адресацією, завантаживши спочатку в базовий регістр зміщення *другого* елемента, а потім у циклі збільшувати значення регістра на 4, просто перестрибуючи елементи з непарними індексами і виконуючи порівняння інших елементів з 0.

За такого методу розв'язання задачі виникає необхідність контролю виходу адреси за межі масиву, який можливий у разі неправильного вибору кількості повторень циклу. Значення лічильника циклу залежить від парності чи непарності кількості елементів у масиві: в масиві з трьох чисел лише один має

парний індекс, і стрибок через третій елемент призводить до виходу за межі масиву, а в масиві з чотирьох елементів уже два мають парні індекси, і такий стрибок потрібно виконувати.

У наведеному нижче прикладі вибрано інший метод адресації елементів масиву – базову індексну адресацію з масштабуванням. По перше, у такому випадку кількість повторень циклу фіксована і визначається тільки кількістю елементів масиву незалежно від того парна вона чи непарна, а по-друге, така адресація дозволяє оперувати не адресами, а логічними індексами 0,1,2,... елементів масиву, що відповідає нашим інтуїтивним уявленням і використовується мовами програмування високого рівня. Недоліком такого підходу, порівняно з методом «перестрибування» непотрібних елементів, є збільшення кількості виконуваних команд.

Суть базової індексної адресації можна сформулювати словосполученням «зміщення від зміщення». Завантаження в базовий регістр адреси початку масиву схоже на перенесення центру координат у нову вихідну точку. Після цього, незалежно від того де розміщується масив у сегменті даних, його елементи адресуються зі зміщення 0 відносно нової точки відліку. Для завдання зміщення «у нових координатах» використовується індексний регістр, який для узгодженості з розміром даних множиться на розмір елемента масиву в байтах (2 для слова). При цьому значення індексного регістра (недарма ж його називають індексним) збільшується в циклі в природному для індексів елементів масиву порядку 0,1,2,3,..., що і дозволяє звертатися до елементів масиву за їх номерами, незалежно від того, які адреси пам’яті насправді їм відповідають.

```
.data
x       dw 1,2,3,-4,-5,6,0,7,8,0,5,8,3
.code
main proc
        lea rbx,x
        xor rax,rax
```

```
        xor rdx,rdx
        xor rsi,rsi
        mov rcx,lengthof x
next:   test rsi,1
        jz @F
        cmp word ptr [rbx+rsi*type x],0
        jle @F
        add ax, word ptr [rbx+rsi*type x]
        adc dx,0
@@:     inc rsi
        loop next
        shl edx,16
        mov dx,ax
        ret
main endp
end
```

Приклад 3.3. Визначення максимального елемента у масиві слів (значення та індексу).

```
.data
x       dw 7,8,0,5,8,3,-10,5,0,0,14,-5,3,-7
.code
main proc
        lea rbx,x
        xor rsi,rsi
        mov ax,[rbx]
        mov rdi,rsi
        mov rcx, lengthof x
        dec rcx
        inc rsi
```

```
next:  cmp [rbx+rsi*2],ax
       jle @F
       mov ax,[rbx+rsi*2]
       mov rdi,rsi
@@: inc rsi
       loop next
       ret
main endp
end
```

Цей приклад дозволить нам також продемонструвати використання групи команд CMOVcc, яку не було розглянуто раніше в межах теми про команди пересилання.

Команди передавання керування погано впливають на роботу конвейера і кеш пам'яті процесора, які використовуються для підвищення його продуктивності. Для зменшення розгалужень у програмах для розв'язування деяких типових задач програмування в систему команд було введено групу команд умовного пересилання.

Узагальнене мнемонічне позначення команд цієї групи – CMOVcc r, r/m. Вони дозволяють виконати завантаження у регістр значення з пам'яті або іншого регістра у разі виконання умови, узагальнене мнемонічне позначення якої «cc» збігається з використовуваним під час опису команд умовного передавання керування. Самі команди не впливають на стан RFLAGS, і тому припустимо використання декількох команд CMOVcc одна за одною.

Приклад 3.2. Визначення значення та індексу максимального елемента у масиві слів з використанням команд умовного пересилання.

```
.data
x      dw 7,8,0,5,8,3,-10,5,0,0,14,-5,3,-7
.code
main proc
```

```
        lea rbx,x
        xor rsi,rsi
        mov ax,[rbx]
        mov rdi,rsi
        mov rcx, lengthof x
        dec rcx
        inc rsi
@@:     cmp word ptr [rbx+rsi*2],ax
        cmovg ax,word ptr [rbx+rsi*2]
        cmovg rdi,rsi
        inc rsi
        loop @B
        ret
main endp
end
```

3.2 Багатовимірні масиви

Як багатовимірні масиви у більшості практичних задач зустрічаються матриці. За класичної інтерпретації матриці як масиву одновимірних масивів обробка матриці виконується за рядками. Для розв'язування деяких задач обробки матриць можна обійтися лінійним циклом, але в більшості випадків використовуються вкладені цикли.

Для адресації елементів матриці використовують базову індексну адресацію з масштабуванням. Для адресації рядків матриці у зовнішньому циклі використовується базовий регістр у який спочатку завантажується адреса початку першого рядка. Подальше переміщення за рядками здійснюється збільшенням значення базового регістра на загальний розмір усіх елементів рядка.

Елементи в рядку адресуються за зміщенням від значення адреси початку рядка в базовому регістрі. Значення зміщення зберігається в індексному регістрі і змінюється у внутрішньому циклі. У разі використання масштабних коефіцієнтів (1, 2, 4, 8, залежно від розміру елемента) значення в індексному регістрі змінюється у внутрішньому циклі від 0 до C-1, де C – кількість елементів рядка (стовпчиків матриці). Така адресація дозволяє оперувати вже не адресами елементів рядка, а логічними індексами 0,1,2,..., що відповідає нашим інтуїтивним уявленням і використовується мовами програмування високого рівня.

Приклад 3.3. Визначення суми елементів кожного рядка матриці. Результати зберігаються в одновимірному масиві result. Перший елемент масиву result – сума елементів першого рядка матриці, другий елемент масиву result – сума елементів другого рядка матриці, і т. д.

```
.data
matrix      dw 1,5,3,2,7
cols  = ($-matrix)/type matrix          ; кількість стовпчиків
            dw 2,5,7,8,9
            dw 4,9,3,2,6
rows  = ($-matrix)/(cols*type matrix)   ; кількість рядків
.data?
result dd rows dup (?)                  ; масив результатів
.code
main proc
      lea rdi,result      ; в rdi – адреса результату внутрішнього циклу
      lea rbx,matrix      ; в rbx – адреса першого рядка
      mov rcx,rows        ; кількість рядків – лічильник зовнішнього циклу
rows_cycle:
      push rcx            ; зберегти лічильник зовнішнього циклу
      xor rax,rax         ; в dx:ax буде накопичуватися сума елементів
```

```
        xor rdx,rdx         ; у внутрішньому циклі
        xor rsi,rsi         ; починаючи з першого (зміщення від бази 0)
        mov rcx,cols        ; для всіх елементів рядка
cols_cycle:
        add ax,word ptr [rbx+rsi*type matrix]       ; додати елемент до суми
        adc dx,0                        ; з урахуванням можливого переносу
        inc rsi                         ; і перейти до наступного
        loop cols_cycle
        shl edx,16              ; скласти з пари dx:ax єдине 32-х бітне
        or dx,ax                ; у регістрі edx
        mov dword ptr [rdi],edx ; і зберегти результат
        add rdi,type result     ; перерахувати адресу наступного результату
        add rbx, cols*type matrix; перерахувати адресу наступного рядка
        pop rcx                 ; і перейти до його обробки
        loop rows_cycle         ; якщо він не останній
        ret
main endp
end
```

Обробка матриці за стовпчикам не значно відрізняється від обробки матриці за рядкам. Змінюється лише порядок вкладення циклів і перерахунок складових адрес. Під час обробки елементів матриці за стовпцями кількість повторень зовнішнього циклу визначається кількістю стовпців матриці (кількістю елементів у рядку), а стовпці адресуються за допомогою базового регістра, початкове значення якого встановлюється рівним адресі початку першого рядка матриці і збільшується на розмір елемента на кожній ітерації зовнішнього циклу.

Кількість повторень внутрішнього циклу визначається кількістю рядків матриці, і рядки адресуються за допомогою індексного регістра внаслідок його

збільшення на загальний розмір елементів рядка від нуля на кожній ітерації внутрішнього циклу.

Приклад 3.4. Визначення суми елементів кожного стовпчика матриці. Результати роботи зберігаються в одновимірному масиві result. Перший елемент масиву result – сума елементів першого стовпчика матриці, другий елемент масиву result – сума елементів другого стовпчика матриці, і т. д.

```
.data
matrix      dw 1,5,3,2,7
cols   = ($-matrix)/type matrix
            dw 2,5,7,8,9
            dw 4,9,3,2,6
rows  = ($-matrix)/(cols*type matrix)
.data?
result      dd cols dup (?)
.code
main proc
    lea rdi,result
    lea rbx,matrix
    mov rcx,cols
cols_cycle:
    push rcx
    xor rax,rax
    xor rdx,rdx
    xor rsi,rsi
    mov rcx,rows
rows_cycle:
    add ax,word ptr [rbx+rsi]
    adc dx,0
    add rsi,cols*type matrix
```

```
        loop rows_cycle
        shl rdx,16
        or rdx,rax
        mov dword ptr [rdi],edx
        add rdi,type result
        add rbx,type matrix
        pop rcx
        loop cols_cycle
        ret
main endp
end
```

3.3 Рядки символів

Рядки символів являють собою звичайні одновимірні масиви, однак через деякі особливості подання їх виділяють в окрему категорію. Рядки символів розглядають як ланцюги байт або слів *довільної довжини*. Ознакою кінця ланцюга байт є нульовий байт (не символ «0» з ASCII-кодом 30h, а двійковий 0), а ознакою кінця ланцюга слів – нульове слово (два нульових байта). Однобайтові рядки містять *коди символів у певному кодуванні*, а рядки слів використовують двобайтове кодування Unicode *з різним порядком зберігання байтів у слові*.

Рядки символів можуть бути оброблені як звичайні одновимірні масиви байтів або слів, однак для розв'язання більшості типових задач зручніше використання команд обробки ланцюжків у поєднанні з префіксами повторення.

Приклад 3.5. Цей приклад демонструє виконання декількох операцій з обробки ASCIIZ-рядків:

1. Визначає довжину рядка.

2. Визначає кількість слів у рядку (словом уважають групу символів, розділених пробілом з кодом 20h)

3. Визначає позицію входження підрядка в рядок.

4. Прибирає пробіли в рядку.

5. Перетворює строкові літери рядка на прописні.

6. Шифрує рядок.

Зробимо попередні зауваження щодо реалізації перетворень символів літер з рядкових на великі та шифрування. Проаналізувавши ASCII таблицю кодувань символів, можна зазначити, що:

1) символи рядкових і символи великих літер займають у ній окремі *неперервні* діапазони;

2) у межах кожного діапазону символи *впорядковано* в алфавітному порядку так, що старшій за алфавітом літері відповідає більше значення ASCII-коду;

3) значення ASCII-кодів рядкових літер відрізняються від ASCII-кодів відповідних їм великих літер на фіксоване значення.

На прикладі ASCII-кодів букв «a» та «A» у двійковій системі не важко побачити, що відрізняються ці коди лише значенням одного біта.

«A» – 41h = 0010 0001b

«a» – 61h = 0**1**10 0001b

Отже, перетворення символу з рядкового на великий виконується скиданням шостого біту байта ASCII-коду, і навпаки, перетворення символу із великого на рядковий виконується встановленням шостого біту байта ASCII-коду.

Найпростіші методи шифрування тексту передбачають виконання над кожним байтом ASCII-коду вихідного повідомлення деякого перетворення, унаслідок якого цей ASCII-код буде змінено.

Таке перетворення може виконуватися з використанням суперпозиції будь-яких арифметичних та/або логічних операцій, але це перетворення *повинно бути зворотнім*, тобто передбачати можливість однозначного поновлення шифрованого тексту.

Властивість зворотності має елементарна логічна функція XOR, для якої в системі команд процесора передбачена однойменна команда.

Наприклад:

0010 0001b XOR 0000 0111b = 0010 0110b – шифрування

0010 0110b XOR 0000 0111b = 0010 0001b – дешифрування

0010 0001b = 41h – «A»

0000 0111b – маска (ключ шифру)

0010 0110b = 46h – «F»

0000 0111b – маска (ключ шифру)

0010 0001b = 41h – «A»

```
.data
str1        db "uiop",0
str2        db "qwertyuiopasdfghjkl",0
str3        db "qw erty uio pasd fgh jkl",0
.code
main proc
; Знайти довжину рядка str3
        lea rdi,str3
        mov rcx,-1
        xor al,al
        repnz scasb
        neg rcx
        lea rcx,[rcx-2]             ; в RCX - довжина рядка str3
; Підрахувати кількість слів у str3
        lea rdi,str3
        xor rbx,rbx
        mov al,20h
@@:  repnz scasb
        jecxz @F
```

```
        inc ebx             ; в EBX – кількість слів
        jmp @B
; Знайти входження рядка str1 у str2
@@:  mov rcx,lengthof str2-lengthof str1+1
        lea rbx,str2
@@:  push rcx
        mov rdi,rbx
        lea rsi,str1
        mov rcx,lengthof str1
        repe cmpsb
        jecxz @F
        inc rbx
        pop rcx
        loop @B
        jmp NotFound
@@:  pop rcx             ; у RBX – адреса початку підрядка str1 у рядку str2
        lea rax,str2
        sub rbx,rax  ; у RBX – позиція (з нуля) підрядка str1 у рядку str2
NotFound:
; Прибрати пробіли в str3
        lea rsi,str3
        mov rdi,rsi
@@:  lodsb
        or al,al
        jz @F
        cmp al,20h
        je @B
        stosb
        jmp @B
```

```
@@: stosb
; Перетворити рядкові літери у str3 на великі
        lea rsi,str3
        mov rdi,rsi
@@: lodsb
        or al,al
        jz @F
        cmp al,"a"
        jb @B
        cmp al,"z"
        ja @B
        and al,11011111b
        stosb
        jmp @B
@@: stosb
; Зашифрувати str3
        lea rsi,str3
        mov rdi,rsi
@@: lodsb
        or al,al
        jz @F
        xor al,10101010b
        stosb
        jmp @B
@@: stosb
        ret
main endp
end
```

3.4 Записи

Запис – це структурований тип даних, що складається з фіксованої кількості елементів довжиною від одного до декількох бітів, які називаються бітовими полями. Для використання записів у програмі потрібно спочатку виконати опис типу (шаблону) запису у вигляді:

Ім’я запису RECORD Ім’я поля:довжина[=значення] [,...]

Імена полів запису повинні бути унікальними, довжина полів зазначається у бітах, за необхідності бітовому полю можна призначити необов’язкове початкове значення ініціалізації за замовчанням. Загальна довжина усіх бітових полів запису не може перевищувати 8/16/32/64 біта.

Наприклад,

```
DATE        RECORD    Year:12, Month:4, Day:5
DATE        RECORD    Year:12=202023, Month:4, Day:5
DATE        RECORD    Year:12=2023, Month:4=1, Day:5=1
```

Опис типу запису – це тільки інформація для транслятора, і пам’ять для його розміщення не виділяється, тому опис типу запису виконується до першої директиви визначення секцій даних і констант.

Після опису типу запису на підставі нього можна визначати одну або декілька змінних у вигляді:

Ім’я змінної Ім’я запису <[значення[,значення]…]>

Ім’я змінної Ім’я запису {[значення[,значення]…]}

Ім’я змінної Ім’я запису кількість DUP (<[значення[,значення]…]>)

Ім’я змінної Ім’я запису кількість DUP ({[значення[,значення]…]})

Ініціалізація бітових полів запису допустима для секції .data, для секції констант .const її можна вважати обов’язковою, а для секції неініціалізованих даних .data? вона недопустима (ігнорується з повідомленням транслятора). Змінні з типом запису у секції неініціалізованих даних .data? визначаються на підставі опису типу без попередньої ініціалізації полів і мають такий вигляд:

.data?

Date1 DATE <>

Date2 DATE {}

Date3 DATE 5 DUP ({})

Date4 DATE 5 DUP (<>)

Змінні з типом запису у секції ініціалізованих даних .data можуть визначатися як на підставі опису типу з попередньою ініціалізацією полів, так і без неї. Для визначення змінної надається можливість прийняти значення попередньої ініціалізації, перевизначити та/або ініціалізувати будь-які поля запису. Символом роздільником полів запису є кома. Якщо ініціалізації підлягають лише деякі бітові поля, то для зазначення їх місцерозташування *зберігають порядок роздільників*. Наприклад:

Date1 DATE 5 dup (<2013,04,24>)

Date2 DATE {2013,,24}

Date3 DATE <2013,04,>

Date4 DATE <2013,04>

Date5 DATE {,,25}

Для змінних з типом запису *виділяється пам'ять*. Розмір наданої для розміщення змінної пам'яті складає 8/16/32/64 біта і визначається *загальною довжиною усіх бітових полів запису* – вибирається найменший розмір достатній для розміщення усіх полів запису. До того ж, якщо загальна довжина усіх бітових полів запису не дорівнює 8/16/32/64 бітам, то ліві (старші) біти байта, слова, подвійні або зчетверені слова не використовуються.

Наприклад, для розміщення змінних з типом DATE, загальна довжина бітових полів якого складає 12+4+5=21 біт, буде надано подвійне слово, старші 11 біт якого будуть зарезервовані. Для підвищення ефективності використання пам'яті в цих бітах можна визначити ще два поля для зберігання часу – hour довжиною 5 бітів і minute довжиною 6 бітів.

Процесор не має команд, які дозволяють працювати з окремими бітовими полями. Обробка окремих бітів або груп бітів може виконуватися тільки у складі

байта, слова, подвійного або зчетвереного слова за допомогою логічних команд, команд зрушення та команд побітової обробки. Зручність у роботі із записами забезпечує транслятор асемблера, який надає такі сервісні послуги:

1) оператор MASK, за допомогою якого можна отримати бітову маску для подальшої локалізації поля запису. Оператор MASK Ім'я поля надає бітову маску, довжина якої збігається з розміром у бітах пам'яті, яка буде виділена для розміщення змінної запису, і в якій усі біти дорівнюють 0, окрім тих бітів, у яких розташовується поле запису із зазначеним ім'ям;

2) оператор WIDTH, за допомогою якого можна визначити розмір у бітах певного поля запису або всього запису в цілому. Оператор WIDTH Ім'я поля надає число, що дорівнює довжині у бітах поля запису із зазначеним ім'ям;

3) особливості трактування транслятором імені полів запису. Ім'я поля запису трактується транслятором як кількість бітів, розташованих справа від цього бітового поля.

Наприклад:

```
.data
Date1        DATE 5 dup (<2023,11,31>)
.....................................................................
    mov eax, Date1          ; в EAX
    mov ebx,eax             ;
    and eax, mask Month     ; and eax,00000000000111100000b
    shr eax, Month          ; або shr eax, width Day ~ shr eax, 5
    cmp eax,12              ; Грудень ?
    jz NewYear              ; Новий рік !
    inc eax                 ; Наступний місяць
    shl eax, Month          ; shl eax, 5
    and ebx, not mask Month; and eax,11111111111111111111111000011111b
    or ebx,eax
    mov Date1,ebx
```

3.5 Структури і об’єднання

Структура – це тип даних, що складається з фіксованої кількості елементів *різних типів*. Для використання структур у програмі потрібно спочатку виконати опис типу (шаблону) структури у вигляді:

Ім’я структури STRUCT [вирівнювання]

Опис першого елемента

……..

Опис останнього елемента

Ім’я структури ENDS

Необов’язкове вирівнювання може зазначатися числом 1, 2, 4, 8, 16, 32. Опис елементів структури виконується так само, як і опис змінних у секції ініціалізованих даних .data, секції неініціалізованих даних .data? і секції констант .const, причому ініціалізовані дані можуть змішуватися з неініціалізованими.

Наприклад,

```
DATE    STRUCT
        Year        WORD 2013
        Month       BYTE ?
        Day         BYTE ?
DATE    ENDS
```

Опис типу структури це тільки інформація для транслятора. Шаблон визначає розміри і, можливо, початкові значення ініціалізації елементів структури, але пам’ять для його розміщення не виділяється, тому опис типу структури виконується до першої директиви визначення секцій даних і констант.

Після опису типу структури на підставі цього можна визначати одну або декілька змінних у вигляді:

Ім’я змінної Ім’я структури <[значення[,значення]…]>

Ім’я змінної Ім’я структури {[значення[,значення]…]}

Ім’я змінної Ім’я структури кількість DUP ({[значення[,значення]…]})

Ініціалізація елементів структури допустима для секції .data, для секції констант .const її можна вважати обов'язковою, а для секції неініціалізованих даних .data? вона недопустима (ігнорується з повідомленням транслятора).

Змінні з типом структури у секції неініціалізованих даних .data? визначаються на підставі опису типу без попередньої ініціалізації полів і мають такий вигляд:

```
.data?
Date1 DATE <>
Date2 DATE {}
Date3 DATE 5 DUP ({})
```

Змінні з типом структури у секції ініціалізованих даних .data можуть визначатися як на підставі опису типу з попередньою ініціалізацією полів, так і без неї. Для визначення змінної надається можливість прийняти значення попередньої ініціалізації, перевизначити та/або ініціалізувати будь-які елементи структури.

Символом роздільником елементів структури є кома. Якщо ініціалізації підлягають лише деякі елементи, то для зазначення їх місцерозташування *зберігають порядок роздільників*. Наприклад:

```
Date1 DATE 5 dup ({2013,04,24})
Date2 DATE {2013,,24}
Date3 DATE <2013,04,>
Date4 DATE <2013,04>
Date5 DATE {,,25}
Date6 DATE {}
```

Для змінних з типом структури виділяється пам'ять. Елементи структури розміщуються в пам'яті послідовно в порядку їх опису в шаблоні структури. Імені змінної з типом структури відповідає базова адреса структури, яка збігається з адресою початку розміщення в пам'яті першого елемента структури.

У разі *використання вирівнювання* розмір фактичної пам’яті, виділеної для розміщення структури, може перевищувати сумарний розмір її елементів, зазначений під час опису шаблону. Це може статися внаслідок того, що елементи структури у пам’яті будуть доповнюватися «зайвими» байтами так, щоб їх зміщення від базової адреси були були кратні вирівнюванню.

Наприклад, для однакових за вмістом структур з різним вирівнюванням їхні елементи будуть розміщені у пам’яті за такими зміщеннями:

```
ALIGNED1        STRUCT    1 ; Вирівнювання 1
element1    WORD      1 ; зміщення = 0
element2    BYTE      2 ; зміщення = 2
element3    DWORD     3 ; зміщення = 3
element4    BYTE      4 ; зміщення = 7
element5    QWORD     5 ; зміщення = 8
element6    BYTE      6 ; зміщення = 16
ALIGNED1 ENDS

ALIGNED2    STRUCT  2   ; Вирівнювання 2
element1    WORD      1       ; зміщення = 0
element2    BYTE      2       ; зміщення = 2 (1 байт додано)
element3    DWORD     3       ; зміщення = 4 ]4/2[= 0
element4    BYTE      4       ; зміщення = 8 ]8/2[= 0
element5    QWORD     5       ; зміщення = 10 ]10/2[= 0
element6    BYTE      6       ; зміщення = 18 (2 байти додано) ]18/2[= 0
ALIGNED2    ENDS
ALIGNED4    STRUCT  4       ; Вирівнювання 4
element1    WORD      1       ; зміщення = 0
element2    BYTE      2       ; зміщення = 2 (1 байт додано)
element3    DWORD     3       ; зміщення = 4 ]4/4[= 0
element4    BYTE      4       ; зміщення = 8 (3 байти додано) ]8/4[= 0
```

```
element5    QWORD    5       ; зміщення = 12 ]12/4[= 0
element6    BYTE     6       ; зміщення = 20 (3 байти додано) ]20/4[= 0
ALIGNED4    ENDS
```

Зручність у роботі зі структурами, незважаючи на вирівнювання, забезпечує транслятор асемблера, який надає оператор «.» (крапка), і директиву PTR.

Оператор «.» (крапка) дозволяє обчислити зміщення до елементів стуктури за їхнім іменем. Директива PTR використовується для зв'язування значення базової адреси структури у регістрі з описом типу структури.

Приклад 3.6. Наступний фрагмент демонструє різні способи організації доступу до елементів структур.

```
DATE    STRUCT
        Year        WORD ?
        Month       BYTE ?
        Day         BYTE ?
DATE    ENDS
.data
Birthday            DATE {1999,09,09}
FamilyBirthday      DATE {1999,09,09},{1977,01,30},{1979,08,15}
.code
main proc
        lea rbx, Birthday           ; у RBX базова адреса структури
; у AX буде значення елемента Year, оскільки він перший описаний у структурі
        mov ax,word ptr [rbx]
;Якщо елемент не перший описаний у структурі, скористаємося оператором «.»
        lea rbx, Birthday.Day
        mov al, byte ptr [rbx]      ; у AL буде значення елемента Day
; Знайдемо елемент масиву FamilyBirthday з найбільшим значенням Year
        lea rbx,FamilyBirthday      ; у RBX базова адреса масиву FamilyBirthday
```

```
        mov ax,(DATE PTR [rbx]).Year        ; у AX початкове значення максимуму
; у RSI зберігаємо адресу елемента масиву FamilyBirthday, у якому міститься
        mov rsi,rbx  ; поточне значення максимального значення
        mov ecx,lengthof FamilyBirthday – 1 ; кількість елементів для обробки
next:   cmp ax, (DATE PTR [rbx+sizeof DATE]).Year ; порівнюємо значення
        ja @F          ; Year наступного елемента з поточним значенням максимуму
        mov ax, (DATE PTR [rbx+sizeof DATE]).Year    ; зберігаємо нове значення
        mov rsi,rbx                                  ; та адресу максимуму
@@:     add rbx,sizeof DATE      ; перехід на наступний елемент масиву
        loop next
        ret
main endp
end
```

Типи елементів структури також можуть бути структурованими, зокрема також бути структурами. Доступ до елементів вкладених структур виконується переліченням розділених крапкою імен полів усіх рівнів вкладеності. У межах одного рівня вкладеності імена елементів структури повинні бути унікальними, тому що вони визначають змішення від базової адреси структури до відповідного елемента.

Наприклад:

```
SNP   STRUCT
        szSurname       BYTE      11     dup(?)
        szName          BYTE      11     dup(?)
        szPatronymic    BYTE      11     dup(?)
SNP ENDS
DATE        RECORD    Year:12, Month:4, Day:5
STUDENT  STRUCT
        StudName        SNP <>
        Burn            DATE <>
```

```
STUDENT ENDS
.data
students      STUDENT <{"Іванов","Іван","Петрович"},{1993,10,4}>,\
                      <{"Петров","Петро","Іванович"},{1993,7,2}>,\
                      <{"Сидоров","Сидір","Петрович"},{1994,12,30}>
.code
main proc
      lea rbx, students
      lea rsi, (STUDENT PTR [rbx]).StudName.szSurname
      lea rdi, (STUDENT PTR [rbx+sizeof STUDENT]).StudName.szSurname
      mov rcx, sizeof SNP.szSurname
      repe cmpsb
      jecxz @F
      .....................................
main endp
end
```

Об’єднання ідентичні структурам, за винятком того, що області об’єднання *перекриваються в пам’яті*, що дозволяє визначати формати різних даних для того самого простору пам'яті. Об'єднання можуть зберігати різні види даних залежно від ситуації.

Водночас кожний елемент у структурі має певне зміщення відносно базової адреси структури, усі елементи об’єднання мають одне і те саме зміщення, тобто ім’я елемента об’єднання позначає різні типи даних залежно від ситуації.

Якщо розмір структури дорівнює сумі розмірів її елементів (із урахуванням вирівнювання), то розмір об’єднання дорівнює довжині найбільшого елемента.

Наприклад обробка елементів об’єднання

LARGE_INTEGER UNION

```
    STRUCT
      LowPart  DWORD ?
      HighPart DWORD ?
    ENDS
  QuadPart QWORD ?
LARGE_INTEGER ENDS
```

може виконуватися по різному в середовищі 32-розрядної та 64-розрядної операційної системи.

```
.data                                .data
Number LARGE_INTEGER <>              Number LARGE_INTEGER <>
.code                                .code
start:                               main proc
    ……….                                 ……….
    mov Number.LowPart,eax               mov Number.QuadPart,rax
    mov Number.HighPart,edx              ……….
    ……….
```

4 ПРОЦЕДУРНЕ ПРОГРАМУВАННЯ

Дуже часто в програмах рід час розв’язання певних типових задач виникає необхідність багаторазового використання абсолютно ідентичних фрагментів програмного коду або таких, що розрізняються незначно. Це збільшує розмір використовуваної програмою пам’яті, зменшує її оглядність і читабельність і просто призводить до появи зайвих помилок.

Для запобігання дублюванню одних і тих самих фрагментів коду, як у межах однієї програми, так і від програми до програми, використовується механізм підпрограм.

Підпрограма являє собою особливим чином оформлену послідовність команд призначену для розв’язання певного завдання, яка має можливість одержувати керування від завдання більш високого рівня та повертати йому керування.

Інакше кажучи, підпрограму можна визначити як правильно оформлену сукупність команд, яка, будучи одноразово описана, за необхідності може бути багато раз використана (викликана) у будь-якому місці програми.

У найпростішому випадку стосовно єдиної підпрограми завданням вищого рівня є вся інша частина програми, яку називають основною або головною програмою. Загалом програма складається з основної програми і декількох підпрограм, і тоді завданням вищого рівня відносно певної підпрограми може бути інша підпрограма.

Загальний розподіл підпрограм на процедури та функції, використовуваний у мовах програмування високого рівня, побудовано на аналізі результатів роботи підпрограми. Якщо підпрограма лише виконує деякі дії – то це процедура, а якщо у результаті виконання підпрограми отримано певне значення – то це функція. Синтаксично асемблер не відрізняє процедури і функції – вони описуються та викликаються абсолютно ідентично.

Механізм підпрограм має більш глибше призначення, ніж просто запобігання дублюванню одних і тих самих фрагментів коду і безпосередньо пов’язаний з концепцією модульного програмування, у якій підпрограма є окремою структурною одиницею. Наразі серед можливостей повного синтаксичного опису підпрограм проаналізуємо лише ті, які дозволяють розглядати підпрограми як один з різновидів команд безумовного передавання управління в програмі, а тему модульного програмування буде розглянуто пізніше.

4.1 Синтаксис оформлення підпрограми

Для опису послідовності команд у вигляді підпрограми в асемблері використовуються дві директиви: PROC и ENDP. Синтаксис оформлення підпрограми:

Ім’я підпрограми PROC

<Пролог>

<Тіло підпрограми>

<Епілог>

RET [число]

Ім’я підпрограми ENDP

<Пролог> містить код ініціалізації для виконання <Тіло підпрограми>. Наприклад, у коді прологу виконується збереження значень, використовуваних у <Тіло підпрограми> регістрів для їх подальшого відновлення, перекладання значень вхідних параметрів в регістри зручні для використання в <Тіло підпрограми> тощо.

<Тіло підпрограми> власне містить набір команд, що реалізують певний алгоритм розв’язання задачі, покладеної на підпрограму.

<Епілог> містить код, що готує вихід з підпрограми. Епілог залежить від коду прологу. Наприклад, якщо в коді прологу зберігалися значення певних регістрів, вони повинні відновлюватися в коді епілогу.

Ім’я підпрограми в директиві PROC визначає точку входу в підпрограму, тобто адресу першої команди підпрограми, що підлягає виконанню після виклику підпрограми.

Підпрограма може розміщуватися в будь-якому місці *секції коду* програми, але так, щоб без виклику на її команди випадково не передалося керування. Якщо підпрограму просто вставити в загальний потік команд у секції коду, то, незважаючи на директиви PROC/ENDP, асемблер сприйматиме команди підпрограми як частину потоку коду.

Підпрограми можна розміщувати в тому самому вихідному файлі, що і головний модуль до початку головного модуля, або після головного модуля, або в іншому файлі, який приєднується до головного модуля директивою INCLUDE.

Наприклад:

Вміст файлу module.asm

```
.code
Third proc
;команди підпрограми
…….....
Third endp
…….
```

Головний модуль

```
inlude module.asm
…….
.code
First   proc
;команди підпрограми
…….....
First   endp
main proc
        ;команди головного модуля
        ……..
        call Third     ;послідовність викликів підпрограм довільна
        ……..
        call Second
        ……..
        call First
        ……..
ret
```

```
Second      proc
;команди підпрограми
…….....
Second      endp
end main
```

4.2 Передавання параметрів у підпрограми

Використання параметрів у підпрограмах збільшує гнучкість їх використання. Параметри в підпрограми можуть передаватися:

1. У регістрах загального призначенняю.
2. У стеку процесора.
3. У регістрах стеку арифметичного співпроцесора.
4. У регістрах розширень MMX/XMM.

Для повернення результатів роботи підпрограма також може використовувати вищезазначені способи.

4.2.1 Передавання параметрів у регістрах загального призначення

Під час передавання параметрів за допомогою регістрів процесора код, що передує виклику підпрограми, розміщує параметри в заздалегідь визначених регістрах і виконує виклик за допомогою команди CALL. Причому регістри для розміщення параметрів бажано вибирати так, щоб код підпрограми зміг відразу використовувати їх без зайвих переміщень у зручні для нього регістри.

Передавання параметрів через регістри є *найкращім варіантом щодо швидкодії*, бо регістрова пам’ять найшвидша, і такий спосіб є найзручнішим для передавання параметрів у розроблені на асемблері підпрограми, але має і певні недоліки:

1) обмежена кількість регістрів загального призначення, унаслідок чого обмежена кількість параметрів;

2) код підпрограми також потребує використання регістрів і за значної кількості переданих у регістрах параметрів буде вимушений зберігати їх у стеку (пам’яті);

3) обмеження на розмір параметрів, що можуть передаватися за значенням, пов’язане з розрядністю регістрів;

4) абсолютна апаратна залежність виклику.

Приклад 4.1. Приклад демонструє програмування з використанням власних підпрограм, аналогічних за своїми функціональними можливостями стандартним функціям мови C/C++. Для передавання параметрів за допогою регістрів виористовуватимемо ті з них, які зручні для коду підпрограми, тобто відзразу можуть використовуватися без зайвого перекладання.

```
.data
src     db "0123456789",0
.data?
dst     db lengthof src dup (?)
.code
strlen proc   ;size_t strlen(const char* String);
;Вхід: RDI – адреса рядка
;Вихід: RAX – довжина рядка
        push rcx      ; зберегти всі
        push rdi      ; використовувані регістри
        mov rcx,-1    ; лічильник повторень для repnz
        xor al,al     ; нульовий байт ознаки завершення рядка
        repnz scasb   ; знайти нульовий байт ознаки завершення рядка
        neg rcx       ; перетворення в прямий код негативного значення RCX
        lea rax,[rcx-2]       ; корекція
        pop rdi       ; поновити використовувані регістри
        pop rcx
        ret
```

```
strlen endp
memcpy      proc   ;void *memcpy(void *dst, const void *src, size_t n)
;Вхід: RDI – адреса приймача, RSI – адреса джерела, RCX – кількість байтів
;Вихід: немає
        push rsi
        push rdi
        push rcx
        push rcx      ; RCX знадобиться два рази
        shr rcx,3     ; ділення RCX на 8 для rep movsq
        rep movsq     ; пересилання
        pop rcx       ; поновити RCX
        and rcx,111b; отримати залишок від ділення RCX на 8
        rep movsb     ; пересилання байтів що залишилися
        pop rcx
        pop rdi
        pop rsi
        ret
memcpy      endp
memset      proc   ;void *memset( void *dest, int c, size_t count );
;Вхід:RDI – адреса рядка, AL – символ, RCX – кількість
;Вихід:      немає
        push rdi
        push rbx
        mov ah,al     ; зібрати
        mov bx,ax     ; в RAX
        shl rax,16    ; 8 байтів
        or ax,bx      ; коду
        mov ebx,eax; символу в AL
        shl rax,32    ; символу в AL
```

```
        or rax,rbx     ; для пересилання
        mov rbx,rcx ; за допомогою rep stosq
        shr ecx,3      ; ділення RCX на 8 для rep stosq
        rep stosq      ; запис в пам'ять
        mov rcx,rbx ; отримати залишок
        and rcx,111b; від ділення RCX на 8
        rep stosb      ; пересилання байтів що залишилися
        pop rbx
        pop rdi
        ret
memset      endp
main proc
        lea rdi,src    ; у RDI адреса початку ASCIIZ-рядка
        call strlen
        mov rcx,rax ; у RAX довжина рядка від strlen
        inc rcx        ; копіюємо і кінцевий нуль
        lea rsi,src    ; у RSI адреса початку джерела
        lea rdi,dst    ; у RDI адреса початку приймача
        call memcpy
        ; адреса початку приймача в RDI не змінилася
        xor al,al      ; в AL байт із значенням заповнювача
        mov rcx,lengthof dst       ; у RCX розмір буфера приймача
        call memset
        ret
main endp
end
```

4.2.2 Передавання параметрів через стек

Під час передавання параметрів через стек процесора код, що передує виклику підпрограми розміщує параметри у стеку *в заздалегідь визначеному порядку* і виконує виклик за допомогою команди CALL.

Команда CALL розміщує на верхівці стеку вслід за параметрами адресу наступної за CALL команди, тобто значення, яке внаслідок повернення з підпрограми командою RET буде значенням регістра RIP.

Для доступу до параметрів необхідно «пірнути» в глибину стеку, не знімаючи з верхівки адресу повернення. Для цього зазвичай використовується базова адресація зі зміщенням з базою в регістрі RBP або RSP (нагадаємо, що у разі використання RBP або RSP як базового регістра за замовчанням виконується звернення до стеку).

Спосіб доступу до параметрів підпрограми вглибині стеку з використанням базової адресації зі зміщенням з базою в регістрі RSP називають Frame Pointer Omission (FPO). Із застосуванням цього способу зміщення для одного і того самого параметра може бути різним у різних місцях коду підпрограми, якщо вона використовує команди, які змінюють значення RSP. Це примушує компілятор контролювати всі такі команди і розраховувати зміщення залежно від поточного стану верхівки стеку.

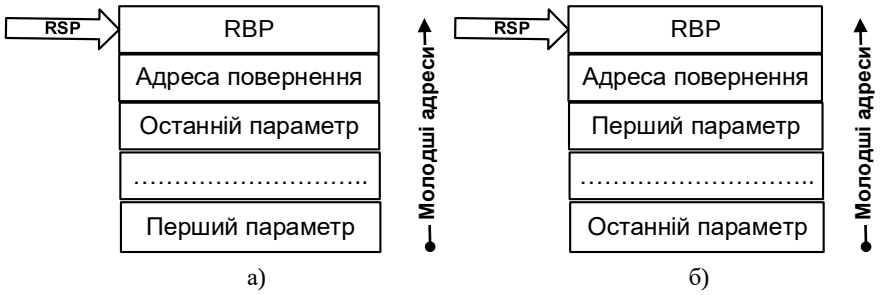
На відміну від FPO, простішим є спосіб доступу до параметрів підпрограми вглибині стеку з використанням базової адресації зі зміщенням з базою в регістрі RBP. Для цього на вході в підпрограму виконується побудова кадру стеку (Stack Frame).

Після збереження в стеку за допомогою PUSH RBP значення RBP і копіювання в нього адреси верхівки стеку за допомогою MOV RBP, RSP залишається лише підрахувати зміщення від RBP до розміщених у стеку параметрів підпрограми. Якщо підпрограма має два і більше параметри, то для

розрахунків зміщенн від RBP потрібно знати порядок (послідовність) у якому код, що викликав підпрограму завантажував параметри у стек.

Компілятори мов програмування високого рівня під час формування коду виклику підпрограм, які отримують параметри через стек, використовують опис підпрограм (функцій) або прототип у яких зазначена кількість і типи параметрів. З огляду на прототип можна визначити два найбільш очевидні порядки запису параметрів підпрограми у стек – так звані прямий і зворотній.

Прямий порядок передбачає послідовне розміщення у стеку параметрів від першого до останнього або зліва-направо, як вони розташовані в описі підпрограми. Зворотній порядок передбачає послідовне розміщення у стеку параметрів від останнього до першого або справа-наліво, як вони розташовані в описі підпрограми. Рисунок 4.1 демонструє зафіксований після виконання команд PUSH RBP і MOV RBP, RSP стековий кадр а) за прямого і б) за зворотнього порядку передавання параметрів.



а) б)

Рисунок 4.2 – Стековий кадр

а) прямий порядок передавання параметрів

б) зворотній порядок передавання параметрів

Отже, у разі при прямого порядку передавання параметрів за адресою [RBP+2*8] буде розташований останній параметр, за адресою [RBP+3*8] буде розташований передостанній параметр, і так далі до першого параметра. У разі

зворотного порядку передавання параметрів [RBP+2*8] буде розташований перший параметр, [RBP+3*8] – другий, і так далі [RBP+i*8] до останнього.

У подальшому значення RBP не змінюється до завершення епілогу підпрограми, що надає можливість використання команд для роботи зі стеком, які змінюють RSP, але при цьому використовуючи RBP можна забирати параметри зі стеку, коли виникне така потреба.

Використовуючи в підпрограмі команди для роботи зі стеком, необхідно забезпечити його стан перед виконанням команди RET таким, яким він був під час входу в підпрограму. Після повернення з підпрограми виконанням команди RET у стеку залишаються параметри, які підпрограма отримувала під час виклику.

Приведення стеку в той самий стан, який він мав *до виклику* підпрограми (так зване вирівнювання) може бути виконано як у коді, що викликав підпрограму, так і безпосередньо у коді самої підпрограми.

Вирівнювання стеку в коді, що викликав підпрограму, виконується за допомогою команди ADD RSP,N після команди CALL. Для вирівнювання стеку безпосередньо у коді самої підпрограми достатньо завершити підпрограму командою RET N. В обох випадках число N дорівнює загальному розміру в байтах усіх параметрів підпрограми у стеку. З урахуванням того, що система команд процесора не дозволяє заносити у стек байт, число N завжди парне.

Передавання параметрів через стек процесора має такі переваги:

1) зменшує апаратну залежність виклику і спрощує реалізацію компіляторів мов високого рівня;

2) майже знімає обмеження на кількість параметрів;

3) знімає обмеження на розмір параметрів, що можуть передаватися за значенням;

4) зменшує прив’язку коду підпрограми до регістрів, оскільки параметри зі стеку можна забирати, коли в цьому виникне така потреба.

Передавання параметрів через стек має і певні недоліки:

1) уповільнює код виклику через необхідність попереднього розміщення параметрів у стеку, який є частиною пам’яті;

2) збільшує код і зменшує швидкість виконання підпрограми через необхідність доступу до параметрів у стеку, який є частиною пам’яті.

Приклад 4.2. Цей приклад відрізняється від попереднього (Приклад 4.1) лише іншою реалізацією передавання параметрів у підпрограми – через стек.

```
.data
src     db "0123456789",0
.data?
dst     db lengthof src dup (?)
.code
strlen proc   ;size_t strlen(const char* String)
;Вхід:адреса рядка у стеку. Вихід:   RAX – довжина рядка
        push rcx
        push rdi
        mov rdi,[rsp+3*8] ; FPO : RDI, RCX, RIP а далі параметр
        mov rcx,-1
        xor al,al
        repnz scasb
        neg rcx
        lea rax,[rcx-2]
        pop rdi
        pop rcx
        ret 8           ; звільняємо стек від покажчика на рядок
strlen endp
memcpy      proc   ;void *memcpy(void *dst, const void *src, size_t n)
        push rbp      ; будуємо
```

```
        mov rbp,rsp ; стековий кадр
        push rdi    ; тепер можна
        push rsi    ; використовувати стек
        mov rdi,[rbp+2*8]; RBP, RIP а далі перший параметр
        mov rsi,[rbp+3*8] ; другий параметр
        mov rcx,[rbp+4*8] ; третій параметр
        shr rcx,3
        rep movsq
        mov rcx,[rbp+4*8] ; знову беремо третій параметр зі стеку
        and rcx,0111b
        rep movsb
        pop rsi
        pop rdi
        pop rbp
        ret     ; стек буде звільняти код що викликав
memcpy  endp
memset  proc   ;void *memset(void *dest, int c, size_t count);
        push rbp
        mov rbp,rsp
        push rdi
        push rbx
        mov rdi,[rbp+3*8+2]     ; параметри
        mov al,[rbp+3*8]        ; передаються
        mov rcx,[rbp+2*8]       ; в прямому порядку
        mov ah,al
        mov bx,ax
        shl rax,16
        or ax,bx
        mov ebx,eax
```

```
        shl rax,32
        or rax,rbx
        shr ecx,3
        rep stosq
        mov rcx,[rbp+2*8]
        and ecx,0111b
        rep stosb
        pop rbx
        pop rdi
        pop rbp
        ret 2*8+2
memset  endp
main proc
        lea rax,src    ; адресу рядка
        push rax       ; розміщуємо в стеку
        call strlen    ; у RAX довжина рядка від strlen
        inc rax        ; будемо копіювати також кінцевий нуль
        push rax
        lea rax,src
        push rax
        lea rax,dst
        push rax
        call memcpy
        add rsp,3*8 ; звільняємо стек від параметрів
        lea rax,dst
        push rax
        xor ax,ax
        push ax
        push lengthof dst
```

```
        call memset
        ret
main endp
end
```

5 МОДУЛЬНЕ ПРОГРАМУВАННЯ

5.1 Концепція модульного програмування

Історія розвитку засобів розробки програм невід'ємно пов'язана з намаганнями максимально спростити і полегшити процес програмування. Особливу актуальність ці завдання набули внаслідок збільшення обсягів і складності програмного забезпечення. Одним з методів розв'язання таких задач є модульне програмування.

Концепція модульного програмування передбачає розподіл складної задачі на простіші функціонально самостійні підзадачі з подальшою реалізацією кожної такої підзадачі або їх окремих груп в окремих програмних одиницях – модулях. Модулі реалізують певну функціональність і пов'язуються між собою попередньо обумовленими вхідними та вихідними даними. Об'єднання окремих модулів у єдиний проєкт виконується спеціальними програмними засобами.

Використання модулів спрощує керування проєктом і надає дві головні переваги.

1. Розробка та налагодження окремих модулів виконується ізольовано від інших модулів проєкту, що дозволяє залучати до його розробки цілі *групи програмістів*.

2. З'являється можливість на кожному з етапів проєкту залучати до розробки модулів різну кількість програмістів *різної кваліфікації*, і до того ж, таких, що володіють *різними мовами програмування*.

Насамперед розглянемо питання використання асемблерних модулів, розроблених з використанням Microsoft Macro Assembler у програмах, розроблених з використанням Microsoft Visual C++, але водночас будуть задіяні

і зворотні зв’язки, оскільки реалізація асемблерних модулів передбачатиме використання сервісних послуг операційної системи Windows NT, а її компоненти написані переважно на C/C++.

Модульне програмування є подальшим продовженням і розширенням процедурного програмування, і окремою структурною одиницею кожного модуля є підпрограма (процедура, функція).

Унаслідок об’єднання розроблених з використанням різних мов програмування окремих модулів у єдиний проєкт їх сумісність забезпечується дотриманням під час розробки підпрограм модулів певних конвенцій виклику та іменування.

5.2 Конвенції виклику

Конвенції виклику (Calling convention) регламентують способи організації виклику підпрограм, передавання їм параметрів і повернення результатів роботи підпрограм у точку виклику. Конвенції виклику визначають такі аспекти.

1. Місце розташування параметрів підпрограми:

а) регістри;

б) стек;

в) регістри та стек;

2. Місце розташування повернених результатів.

3. Порядок передавання параметрів:

а) із використанням регістрів – зіставлення регістрів з номерами параметрів в описі підпрограми;

б) із використанням стека – прямий (зліва направо) або зворотний (справа наліво) порядок зіставлення параметрів в описі підпрограми;

4. Із використанням для передавання параметрів стека – хто вирівнює стек після завершення виклику:

а) підпрограма;

б) код, що викликав підпрограму.

5. Вміст яких регістрів підлягає обов'язковому збереженню і поновленню в підпрограмі.

Протягом розвитку програмування для 32-розрядних версій Windows NT сформувалося декілька конвенцій виклику, серед яких можна зазначити такі:

1. PASCAL – параметри передаються через стек у прямому порядку (зліва направо), стек вирівнює підпрограма.

2. C (CDECL) і SYSCALL – параметри передаються через стек у зворотному порядку (справа наліво), стек вирівнює код, що викликає підпрограму.

3. STDCALL – параметри передаються через стек у зворотному порядку (справа наліво), стек вирівнює підпрограма.

4. FASTCALL (REGISTER) – нестандартизована і існує в декількох реалізаціях:

а) у реалізації Embarcadero C++ Builder для 32-розрядних версій Windows NT: перший (зліва направо) параметр у регістрі EAX, другий – в EDX, третій – в ECX, інші, за наявності таких, розміщуються в стеку в прямому порядку (зліва направо), стек вирівнює підпрограма;

б) у реалізації Microsoft Visual C++ для 32-розрядних версій Windows NT: перший (зліва направо) параметр у регістрі ECX, другий – в EDX, інші, за наявності таких, розміщуються у стеку в зворотному порядку (справа наліво), стек вирівнює підпрограма.

Конвенції CDECL, STDCALL і FASTCALL для *32-розрядних* версій Windows NT передбачають повернення результату з підпрограми через регістр-акумулятор за значенням в EAX/AX/AL, якщо його розмір не перевищує 32 біти. Результат розміром 8 байтів повертається за значенням у парі регістрів EDX:EAX, усе інше повертається за адресою в EAX. Числа з плаваючою комою повертаються за значенням через верхівку стека арифметичного співпроцесора.

Усі 64-розрядні версії Windows NT використовують єдиний ABI (англ. Application Binary Interface, ABI) (Рисунок 5.1).

1. Перші чотири зліва направо параметри передаються через регістри, п’ятий та інші параметри, за наявності таких, розміщуються у стеку в зворотному порядку (справа наліво, тобто від останнього до п’ятого)

2. Для параметрів розміром 8/16/32/64 біта – 1-й (зліва направо) у RCX, 2-й – у RDX, 3-й – у R8, 4-й – у R9.

func1(int a, int b, int c, int d, int e);

// a – RCX, b – RDX, c – R8, d – R9, e – у стеку

3. Якщо перші чотири параметри є числами з плаваючою комою, то вони передаються зліва направо в регістрах XMM0–XMM3.

func2(float a, double b, float c, double d, float e);

// a – XMM0, b – XMM1, c – XMM2, d – XMM3, e – у стеку

4. Якщо перші чотири параметри різнотипні, то вони передаються зі збереженням закріплених регістрів відповідно до пунктів 2 і 3.

func3(int a, double b, int c, float d);

// a – RCX, b – XMM1, c – R8, d – XMM3

5. Усі параметри у стеку мають розмір вісім байт, незалежно від їх фактичного розміру.

6. Код, що викликає підпрограму, резервує у стеку місце для перших чотирьох параметрів, хоча і передає їх у регістрах (Рисунок 5.1). Зберігати параметри в цій області не обов’язково, і її можна використовувати для зберігання регістрів, які змінюються кодом підпрограми.

7. Стек вирівнює код, що викликає підпрограму.

8. RDI, RSI, RBX, RBP, R12–R15, XMM6–XMM15 повинні зберігатися підпрограмою.

9. Результат, що не перевищує 64 біта, повертається в RAX за винятком чисел з плаваючою комою, які повертаються в XMM0. Результат, що перевищує 64 біта повертається за посиланням у RAX.

Різноманіття конвенцій викликів для 32-розрядних Windows NT створювало певні незручності. Завдяки розвитку 64-розрядних Windows NT з

цим різноманіттям врешті решт покінчено. Якщо раніше Microsoft Visual C++ внутрішньо використовував для 32-розрядних додатків __cdecl або __fastcall, то зараз навіть для внутрішніх функцій для 64-розрядних додатків використовується ABI.

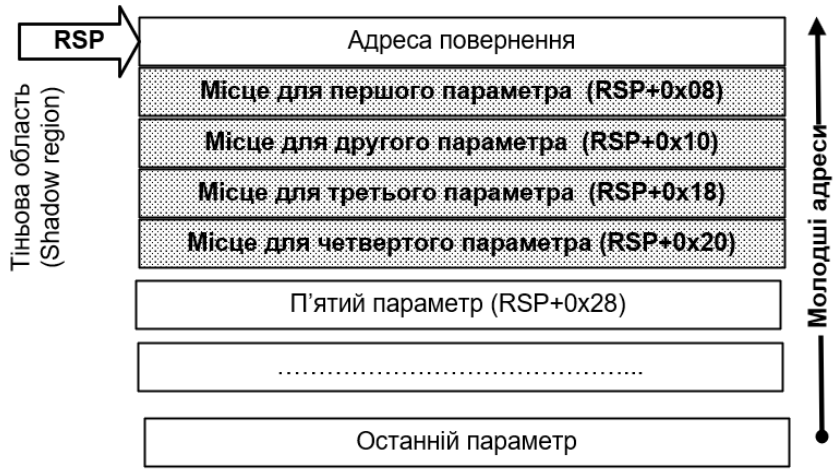


Рисунок 5.1 – Резервування місця в пам'яті для параметрів

5.3 Підпрограма як структурна одиниця модуля

Для асемблера мовним засобом є синтаксичний опис підпрограми, який має такий вигляд:

ім'я PROC [<видимість>] [USES <Регістри>] [FRAME [:адреса обробника]] [LOCAL <Змінні>]

<послідовність команд>

RET

ім'я ENDP

У квадратних дужках «[…]» зазначено необов'язкові директиви. Атрибут <видимість> визначає, чи буде підпрограма доступна (PUBLIC та EXPORT) для виклику з інших програм або модулів, чи не буде (PRIVATE).

<Регістри> – визначає для директиви USES перелік розділених пробілом імен регістрів загального призначення, які будуть автоматично зберігатися на вході у підпрограму і поновлюватися на виході.

FRAME [:адреса обробника] змушує MASM генерувати запис таблиці функцій у секції .pdata програмного файла та розгортати інформацію у секції .xdata для структурної обробки виключень і призначити адресу обробника виключення.

Директива LOCAL дозволяє визначити локальні змінні для використання в підпрограмі. Пам'ять для локальних змінних резервується у стеку підпрограми і область їх видимості – тільки код підпрограми. <Змінні> в директиві LOCAL являють собою розділений комами перелік, що має синтаксис

Змінна[кількість]:тип [, ……]

Тобто перелік складається з елементів із синтаксисом двох типів:

Змінна:тип – для змінних простих і структурованих типів (але не масивів).

Змінна[кількість]:тип – для масивів з елементами простих і структурованих типів.

Змінна – унікальне у межах підпрограми символьне ім'я, а тип – будь-який простий тип або попередньо визначений структурований тип.

У разі великої кількості локальних змінних для перенесення переліку змінних на наступний рядок використовується символ «\» після коми або декілька директив LOCAL.

5.4 Статичні бібліотеки

5.4.1 Структура вихідного коду статичних бібліотек

Каркасний код статичної бібліотеки має такий вигляд:

```
.code
<визначення підпрограм>
…….
```

end

Усі призначені для використання поза межами модуля (експортовані) функції статичної бібліотеки повинні визначатися з атрибутом видимості PUBLIC.

Трансляція вихідного тексту майбутньої бібліотеки виконується звичайним способом – ML64.EXE /c /coff /Cp /nologo ім'я.asm.

Подальше отримання бібліотечного lib-файлу виконується за допомогою спеціального параметра лінкера /LIB – LINK.EXE /LIB ім'я.obj або з використанням спеціальної утиліти Microsoft Library Manager LIB.EXE ім'я.obj.

Приклад 5.1 Статична бібліотека slldemo.lib у якій реалізовано альтернативні та розширені варіанти стандартних функцій С для обробки рядків. Зокрема стандартна С-функція strcat вміє об'єднувати тільки два рядки, а її розширений бібліотечний аналог astrcat – довільну кількість.

```
; Файл slldemo.asm
.code
StringLength        proc PRIVATE
; Внутрішня функція для визначення довжини ASCIIZ-рядка
; Передавання параметрів регістрове
; Вхід: RDI - адреса рядка, Вихід: RAX - довжина
; Внутрішня функція зберігає ВСІ регістри, які використовує
        push rdi
        push rcx
        mov rcx,-1
        xor al,al
        repnz scasb
        neg ecx
        lea rax,[ecx-2]
        pop rcx
        pop rdi
```

```
        ret
StringLength endp
option prologue:none      ; ці директиви забороняють транслятору самостійно
option epilogue:none      ; генерувати код прологу та епілогу які порушують ABI
astrlen proc PUBLIC       ;size_t strlen(const char* String)
        mov [rsp+8],rdi
        mov rdi,rcx
        call StringLength
        mov rdi,[rsp+8]
        ret
astrlen endp
amemcpy proc PUBLIC ;void *memcpy(void *dst, const void *src, size_t n)
        mov [rsp+8],rsi
        mov [rsp+16],rdi
        mov [rsp+24],rbx
        mov rdi,rcx
        mov rsi,rdx
        mov rcx,r8
        mov rbx,rcx
        shr rcx,3
        rep movsq
        mov rcx,rbx
        and rcx,7
        rep movsb
        mov rsi,[rsp+8]
        mov rdi,[rsp+16]
        mov rbx,[rsp+24]
        ret
amemcpy endp
```

```
astrcpy        proc PUBLIC ;char *strcpy(char *strDestination, const char *strSource)
        mov [rsp+8],rsi
        mov [rsp+16],rdi
        mov rdi,rdx
        call StringLength
        mov rsi,rdi
        mov rdi,rcx
        lea rcx,[rax+1]      ; копіюємо і 0 для завершення рядка
        shr ecx,3
        rep movsq
        lea rcx,[rax+1]
        and ecx,7
        rep movsb
        mov rsi,[rsp+8]
        mov rdi,[rsp+16]
        ret
astrcpy endp
amemset      proc PUBLIC ;void *memset( void *dest, unsigned char c, size_t count );
        mov [rsp+8],rdi
        mov [rsp+16],rbx
        mov rdi,rcx
        mov rax,rdx
        mov ah,al
        mov bx,ax
        shl rax,16
        or ax,bx
        mov ebx,eax
        shl rax,32
        or rax,rbx
```

```
        mov rcx,r8
        shr rcx,3
        rep stosq
        mov rcx,r8
        and rcx,7
        rep stosb
        mov rdi,[rsp+8]
        mov rbx,[rsp+16]
        ret
amemset     endp
cat2str     proc PRIVATE
        call StringLength
        mov rcx,rax
        xchg rsi,rdi
        rep movsb
        xchg rsi,rdi
        ret
cat2str endp

astrcat     proc PUBLIC ; astrcat(unsigned int strcount, ...)
; Виконує конкатенацію декількох ASCIIZ-рядків
; Вхід: strcount – кількість рядків, що буде приєднуватися
; ... – перелік адрес рядків. До першого рядка будуть приєднуватися усі інші
; Вихід: немає
        mov [rsp+8],rsi
        mov [rsp+16],rdi
        mov [rsp+24],rbx
        mov [rsp+32],rbp
        mov rbx,rcx
```

```
        cmp rbx,2    ; параметрів функції повинно
        jae @F       ; бути не менше двох
        ret          ; недостатня кількість параметрів функції
@@:     mov rdi,rdx
        call StringLength
        lea rsi,[rdi+rax]    ; RSI – адреса кінця рядка приймача
        cmp rbx,2
        jne @F
        mov rdi,r8
        call cat2str
        jmp exit
@@:     cmp rbx,3
        jne @F
        mov rdi,r8
        call cat2str
        mov rdi,r9
        call cat2str
        jmp exit
@@:     mov rdi,r8
        call cat2str
        mov rdi,r9
        call cat2str
        mov r10,(1+3)*8   ; return address + 3 shadows
        sub rbx,3
@@:     or rbx,rbx
        jz exit
        add r10,8         ; + 1 shadow the first time and the next parameter after
        mov rdi,[rsp+r10]
        call cat2str
```

```
        dec rbx
        jmp @B
exit:   xor al,al
        mov rdi,rsi
        stosb
        mov rsi,[rsp+8]
        mov rdi,[rsp+16]
        mov rbx,[rsp+24]
        mov rbp,[rsp+32]
        ret
astrcat endp
end
```

5.4.2 Зв’язування MASM зі статичними бібліотеками

Для зв’язування проєкту Microsoft Macro Assembler зі статичною бібліотекою потрібно виконати опис використовуваних функцій бібліотеки у вихідному тексті асемблерної програми, або у використовуваних нею заголовних файлах у вигляді EXTERNDEF ім’я:PROC.

Після цього потрібно додати ім’я файлу бібліотеки в командний рядок лінкера або скористатися директивою INCLUDELIB. Під час лінкування проєкту файл бібліотеки повинен розміщуватися в каталозі проєкту, або в стандартних каталогах, у яких лінкер виконує пошук бібліотек, або зазначатися за повним ім’ям.

Приклад 5.2 Додаток на асемблері slldemoasmcaller.exe, що демонструє використання бібліотеки slldemo.lib (Приклад 5.1).

```
; Файл slldemoasmcaller.asm
include \masm64\include64\masm64rt.inc
includelib slldemo.lib
externdef astrlen:proc
```

```
externdef astrcpy:proc
externdef astrcat:proc
externdef amemset:proc
externdef amemcpy:proc
.data
str1        CHAR "First string",0
str2        CHAR "+second string",0
str3        CHAR "+third string",0
str4        CHAR "+fourth string",0
str5        CHAR "+fifth string",0
str6        CHAR "+sixth string",0
.data?
buffer      db 256 dup (?)
.code
main        proc
        call AllocConsole
        lea rsi,str1
        lea rdi,buffer
        mov rcx,rdi
        mov rdx,rsi
        call astrcpy
        mov rcx,cfm$("\nastrcpy:\t\t")
        call StdOut
        lea rcx,buffer
        call StdOut
        mov rcx,6
        lea rdx,buffer
        lea r8,str2
        lea r9,str3
```

```
lea rax,str4
mov [rsp+4*8],rax
lea rax,str5
mov [rsp+5*8],rax
lea rax,str6
mov [rsp+6*8],rax
call astrcat
add rsp,(1+4+3)*8
mov rcx,cfm$("\nastrcat:\t\t")
call StdOut
mov rcx,rdi
call StdOut
mov rcx,rdi
mov rdx,"x"
mov r8,15
call amemset
mov rcx,cfm$("\namemset:\t\t")
call StdOut
mov rcx,rdi
call StdOut
mov rcx,rsi
call astrlen
mov rcx,rdi
mov rdx,rsi
mov r8,rax
call amemcpy
mov rcx,cfm$("\namemcpy:\t\t")
call StdOut
mov rcx,rdi
```

```
        call StdOut
        mov rcx,rsi
        call astrlen
        mov rbx,rax
        rcall vc_printf,cfm$("%s%s%s"),cfm$("\nastrlen:\t\tLength of string \l"),rsi
        rcall vc_printf,cfm$("%s%d"),cfm$("\r equal "),rbx
        waitkey cfm$("\nPress \lEnter\r ...")
        ret
main endp
end
```

5.4.3 Зв’язування Microsoft Visual C++ зі статичними бібліотеками

Для зв’язування проєкту Microsoft Visual C++ зі статичною бібліотекою потрібно виконати опис прототипів використовуваних функцій бібліотеки у вихідному тексті С-програми у вигляді:

extern "C" <тип_результату> ім’я(тип_параметрів);

Директива extern з модифікатором "C" повідомляє компілятор про те, що це імпортована функція і шукати її потрібно в бібліотечних файлах.

Після цього потрібно додати ім’я файлу бібліотеки в командний рядок лінкера в настройках властивостей проєкту або скористатися директивою #pragma comment(linker, "/DEFAULTLIB:ім’я.lib") у вихідному тексті програми.

Під час лінкування проєкту файл бібліотеки повинен розміщуватися в каталозі проекту, або в стандартних каталогах, у яких лінкер виконує пошук бібліотек, або зазначатися за повним ім’ям. Після цього виклик функцій виконується звичним для C\C++ способом.

Приклад 5.3. Додаток мовою C++, що демонструє використання бібліотеки slldemo.lib (Приклад 5.1).

```
; Файл slldemocppcaller.cpp
#include <iostream>
```

```
#include <cstdlib>
#include <cstdio>
using namespace std;
extern "C" {
   unsigned int astrlen(char* lpstr);
   void amemcpy(char* lpdst, char* lpsrc, unsigned int n);
   void astrcpy(char* lpdst, char* lpsrc);
   void amemset(char* lpdest, char chr, unsigned int count);
   void astrcat(unsigned int strcount, ...);
}
int main()
{
        char str1[] = "First string";
        char str2[] = "+second string";
        char str3[] = "+third string";
        char str4[] = "+fourth string";
        char str5[] = "+fifth string";
        char str6[] = "+sixth string";
        char buffer[256];
        astrcpy(buffer, str1);
        cout << "astrcpy:\t" << buffer << endl;
        astrcat(6, buffer, str2, str3, str4, str5, str6);
        cout << "astrcat:\t" << buffer << endl;
        amemset(buffer, 'x', 15);
        cout << "amemset:\t" << buffer << endl;
        amemcpy(buffer, str1, astrlen(str1));
        cout << "amemcpy:\t" << buffer << endl;
        cout << "astrlen:\tLength of string '" << str1 << "' equal " << astrlen(str1) <<
endl;
```

```
    system("pause");
    return 0;
}
```

5.5 Динамічні бібліотеки

5.5.1 Концепція динамічних бібліотек

Під час компонування зі статичною бібліотекою лінкер шукає код потрібної функції в усіх lib- і obj-файлах, зазначених у командному рядку та/або відповідними директивами, а, знайшовши, *копіює код потрібної функції* з цих файлів і *вставляє* його в компонований ним кінцевий програмний файл. Якщо функція використовує інші функції цієї або іншої бібліотеки, їх код також буде скопійовано і уставлено в кінцевий програмний файл, якщо ці функції водночас залежать від інших функцій – їх код також буде скопійовано і т. д.

Отже, унаслідок використання статичних бібліотек програмні файли двох різних додатків, що використовують одну і ту саму функцію однієї бібліотеки, містимуть програмний код цієї функції і тих, від яких вона залежить. Як наслідок, програмні файли «важчають» на розмір коду використаних функцій, і після завантаження додатка на виконання та створення процесу в пам’яті міститимуться однакові фрагменти коду.

Подальшим розвитком ідей модульності є динамічні бібліотеки (англ. Dynamic Link Library, DLL). Спочатку передбачалося, що DLL дозволять зменшити витрати системних ресурсів завдяки використанню лише одного екземпляра DLL на диску і в пам’яті різними додатками, а згодом концепція значно розширилися.

5.5.2 Методи зв’язування додатка з динамічними бібліотеками

Для того, щоб програма могла викликати ту або іншу функцію, DLL, що містить її код, повинна бути завантажена в адресний простір процесу. Коли саме DLL буде завантажена в пам’ять процесу, залежить від використовуваного

додатком методу зв’язування – раннього (статичного лінкування) або пізнього (динамічного лінкування).

Як для раннього, так і для пізнього зв'язування використовується спеціальна структура – таблиця експорту (англ. Export Address Table, EAT), яка є в кожної DLL. У EAT перелічені експортовані DLL функції, їхні номери (ордінали) і відносні адреси функцій у межах DLL – RVA (англ. Relocate Virtual Address). RVA за відомої базової адреси завантаження DLL в адресний простір процесу дозволяють обчислити точку входу в будь-яку функцію DLL.

5.5.2.1 Раннє зв’язування

Під час компонування програми, що використовує раннє зв’язування з DLL, лінкеру доводиться працювати з віртуальними функціями, тобто такими, які взагалі існують, але на момент лінкування їхнього коду немає у створюваному програмному файлі і в пам’яті він з’явиться лише внаслідок завантаження цього самого програмного файлу, що потрібно якось створити.

Для раннього зв’язування лінкер формує так звану таблицю імпорту програмного файлу (англ. Import Address Tables, IAT). Таблиця імпорту – це особлива структура (її місце розташування зазначається в PE-заголовку програмного файлу), що містить список використовуваних програмою DLL і список імпортованих з кожної DLL функцій. Для лінкування додатка програмні файли DLL не потрібні і для створення IAT використовуються бібліотеки імпорту. Бібліотеки імпорту – це спеціальні файли з розширенням .lib, які містять усю потрібну для лінкера інформацію про DLL, які має використати додаток.

Для кожної функції в таблиці імпорту є елемент для зберігання адреси крапки входу в функцію. Хоча на стадії лінкування ця адреса невідома, але це не заважає лінкеру як операнди команд переходу використати непряму адресацію по відповідних елементах таблиці імпорту, адреси яких він може обчислити.

У процесі завантаження програмного файлу на виконання завантажник Windows аналізує його таблицю імпорту, визначає, які DLL використовує

програма, знаходить їх, завантажує в адресний простір процесу й заносить в таблицю імпорту реальні адреси функцій цих DLL. Рисунок 5.2 демонструє процес раннього зв’язування.

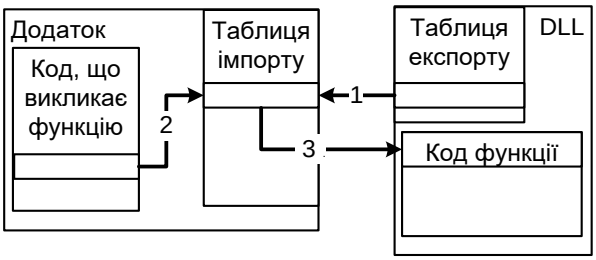


Рисунок 5.2 – Схема процесу раннього зв’язування

Під час підготовки додатка до виконання завантажувач аналізує його таблицю імпорту і визначає всі використовувані додатком DLL і функції цих DLL. Потім відбувається пошук і завантаження потрібних DLL в адресний простір додатка, пошук у таблиці експорту DLL адрес використовуваних додатком функцій і заповнення адрес у таблиці імпорту додатка (1). У момент виклику функції з таблиці імпорту береться адреса функції (2) і відбувається власне виклик функції (3).

У разі раннього зв’язування на момент запуску додатка всі необхідні DLL виявляються завантаженими, таблиця імпорту заповнена – і все це система иконує вавтоматично, без участі самого додатка, але відсутність у процесі завантаження додатка зазначеної в його таблиці імпорту DLL призведе до аварійного припинення завантаження. Окрім того, дуже часто немає необхідності завантажувати всі використовувані програмою DLL у момент її запуску.

5.5.2.2 Пізнє зв’язування

Метод пізнього зв’язування відрізняється від раннього зв’язування тим, що завантаження DLL і визначення адреси потрібної функції виконується самим додатком у процесі його роботи за допомогою функцій Win API LoadLibrary

(LoadLibraryEx) і GetProcAddress. Ці функції перебувають у kernel32.dll, і з якою попередньо необхідно виконати статичне (раннє) лінкування.

У будь-який момент часу додаток може виконати пізнє зв’язування за схемою, наведеною на рисунку 5.3.

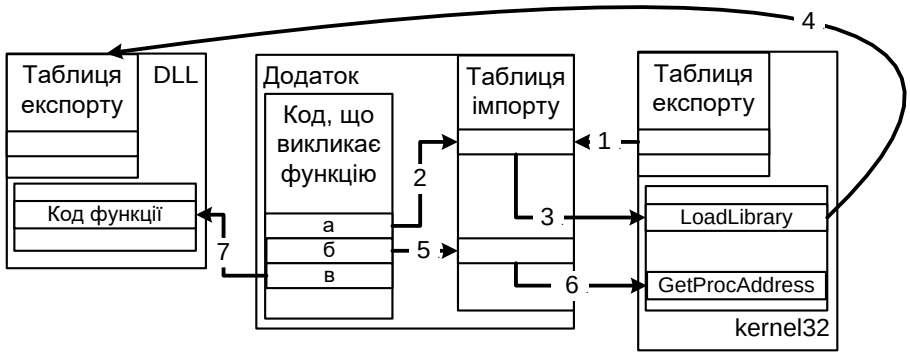


Рисунок 5.3 – Схема процесу пізнього зв’язування

Для пізнього зв’язування потрібно виконати такі дії:

1. За допомогою Win API функції LoadLibrary/LoadLibraryEx завантажити потрібну DLL.

2. За допомогою Win API функції GetProcAddress одержати адресу потрібної функції в завантаженій DLL.

3. Викликати функцію, скориставшись одержаною у попередньому пункті адресою.

4. За допомогою Win API функції FreeLibrary вивантажити DLL, що вже непотрібна, водночас звільняючи, таким чином, пам’ять.

Напрямок 2–3–4 (Рисунок 5.3) відповідає завантаженню бібліотеки за допомогою LoadLibrary, а 5–6 – визначенню адреси потрібної функції за допомогою GetProcAddress. Потім можна викликати функцію DLL (7), і звісно, для цього таблиця імпорту непотрібна. Щоб не викликати знову і знову GetProcAddress перед кожним викликом функції з DLL, додаток може

однократно визначити адреси функцій, що його цікавлять, і зберегти їх у масиві або деяких змінних.

5.5.2.3 Порядок пошуку DLL під час зв'язування

Для раннього зв'язування в таблиці імпорту додатка ім'я DLL зазначається без задання шляху, а для пізнього зв'язуванні у відповідному параметрі функцій LoadLibrary(LoadLibraryEx) також *можна* зазначати тільки ім'я DLL. Це робить додатки більш «мобільними», тобто не прив'язаними до певного місця розташування у структурі каталогів, яке на різних системах може бути різним.

Пошук файлу з DLL під час зв'язування виконується за певними заздалегідь визначеними правилами. Порядок пошуку DLL у Windows NT залежить від того, чи використовується в системі так званий режим безпечного пошуку DLL (SafeDllSearchMode). Якщо цей режим увімкнутий, то пошук DLL виконується в такій послідовності:

1. Поточний каталог додатка.
2. Системний каталог Windows (%SystemRoot%\System32).
3. 16-розрядний системний каталог (%SystemRoot%\System – застаріло).
4. Каталог установлення Windows (%SystemRoot%).
5. Поточний каталог користувача.
6. Каталоги, перелічені в змінній оточення %PATH%.

Ще починаючи з Windows XP SP2 режим безпечного пошуку DLL використовується за замовчуванням. Для того щоб вимкнути цей режим, потрібно створити в розділі реєстру HKLM \ System \ CurrentControlSet \ Control \ Session Manager DWORD-параметр SafeDllSearchMode зі значенням 0 (значення 1 знову вмикає). Якщо SafeDllSearchMode вимкнути, то пошук DLL буде виконуватися в такій послідовності:

1. Поточний каталог додатка.
2. Поточний каталог користувача.
3. Системний каталог Windows (%SystemRoot%\System32).

4. 16-розрядний системний каталог (%SystemRoot%\System – застаріло).
5. Каталог встановлення Windows (%SystemRoot%).
6. Каталоги, перелічені в змінній оточення PATH.

5.5.3 Розробка динамічних бібліотек

5.5.3.1 Структура вихідного коду динамічних бібліотек

Каркасний код DLL має такий вигляд:

```
.code
DllMain      proc
        <код ініціалізації>
        mov rax, TRUE
        ret
DllMain endp
<визначення підпрограм>
end
```

DLL являє собою програмний файл PE-формату і може мати такі самі секції, як і звичайні додатки. Зокрема DLL може експортувати тільки дані та/або ресурси і зовсім не експортувати функції. DLL має вхідну точку, однак не може виконуватися самостійно – *DLL завжди виконується у контексті певного потоку* певного процесу.

Унаслідок використання DLL двома і більше додатками її секції коду і даних є поділюваними – вони відображаються у адресний простір кожного додатка і спільно ними використовуються.

Фактично для секцій DLL операційна система використовує так званий механізм копіювання для запису (англ. Copy-On-Write, COW). Цей механізм передбачає спільне використання секцій DLL декількома додатками до того моменту, доки будь-який з них не спробує виконати запис у поділювану пам’ять. У цьому випадку в адресному просторі додатка створюється його власна копія поділюваної раніше пам’яті, у яку виконується операція запису. Інші додатки

продовжують спільно використовувати секції DLL, доки будь-який з них також не спробує виконати в поділювану пам'ять запис, і, унаслідок цього також не отримає її копію для власних потреб.

Ураховуючи, що для секцій даних, зазвичай, притаманна операція запису, в адресному просторі кожного додатка, який використовує DLL створюється копія секції даних DLL. *Власного стека DLL не має – її код завжди використовує стек потоку, у контексті якого він виконується*.

Вхідної точкою DLL є функція з прототипом

```
BOOL WINAPI DllMain(__in  HINSTANCE hinstDLL,
                    __in  DWORD fdwReason,
                    __in  LPVOID lpvReserved );
```

Функція DllMain викликається системою за певних подій і її призначення полягає в ініціалізації DLL, тобто підготовці її до подальшого використання. Під час виклику функція DllMain отримує від системи три параметри.

Через параметр hinstDLL система повідомляє DLL її власний ідентифікатор (хендл, маніпулятор, дескриптор). За необхідності в DLL можна використовувати значення hinstDLL у функціях, параметром яких є ідентифікатор модуля. У Windows ідентифікатор DLL дорівнює базовій адресі, за якою завантажена DLL.

Параметр fdwReason спільно з параметром lpvReserved визначають причини, за яких відбувся виклик DllMain.

Якщо параметр fdwReason дорівнює DLL_PROCESS_ATTACH, а lpvReserved не дорівнює нулю, то бібліотека завантажується в адресний простір процесу за статичного (раннього) зв'язування.

Якщо параметр fdwReason дорівнює DLL_PROCESS_ATTACH, а lpvReserved дорівнює нулю, то бібліотека завантажується в адресний простір процесу за динамічного (пізнього) зв'язування в результаті виклику функції LoadLibrary/LoadLibraryEx.

Якщо fdwReason дорівнює DLL_PROCESS_DETACH, а lpvReserved не дорівнює нулю, то бібліотека вивантажується з адресного простору процесу внаслідок завершення процесу.

Якщо fdwReason дорівнює DLL_PROCESS_DETACH, а lpvReserved дорівнює нулю, то бібліотека вивантажується з адресного простору процесу внаслідок виклику функції FreeLibrary.

Значення fdwReason рівне DLL_THREAD_ATTACH означає, що поточний процес створив новий потік і система викликає в контексті цього потоку функції DllMain усіх DLL, завантажених в адресний простір процесу.

Значення fdwReason рівне DLL_THREAD_DETACH означає, що в процесі завершується деякий потік і система викликає в контексті цього потоку функції DllMain усіх DLL, завантажених в адресний простір процесу.

Коли система викликає функцію DllMain з параметром fdwReason, що дорівнює DLL_PROCESS_ATTACH, вона повинна повернути системі код завершення, який визначає подальшу долю DLL. Якщо повертається FALSE під час ініціалізації процесу, який використовує раннє зв'язування з DLL – процес завершується з помилкою. Якщо повертається FALSE з використанням процесом пізнього зв'язування – функція LoadLibrary повертає нульове значення, система викликає функцію DllMain з параметром fdwReason, що дорівнює DLL_PROCESS_DETTACH і вивантажує DLL. В усіх інших випадках код повернення DllMain ігнорується.

Усі призначені для використання поза межами DLL (експортовані) функції динамічної бібліотеки повинні визначатися з атрибутом видимості EXPORT. Також керувати створенням таблиці експорту можна за допомогою параметрів командного рядка лінкера, однак найбільше можливостей щодо керування експортом для побудови бібліотек DLL надають файли визначення модуля.

5.5.3.2 Файли визначення модуля

Файли визначення модуля являють собою звичайні текстові файли з розширенням *.def, які надають лінкеру додаткову інформацію щодо особливостей компонування модуля. Типова структура def-файлу DLL має такий вигляд:

LIBRARY [library] [BASE=address]
NAME [application]
EXPORTS
entryname[=internalname] [@ordinal [NONAME]] [PRIVATE] [DATA]

entryname – це ім'я функції або змінної, за яким вона буде експортуватися. Це ім'я функції або змінної *може бути не визначене в DLL*. Можна експортувати за іменем, що відрізняється від імені функції або змінної в DLL виконавши зіставлення entryname зі справжнім визначеним у DLL іменем internalname.

@ordinal дозволяє вказати порядковий номер (ординал), за яким експортується ім'я entryname. Цей номер потрапить у таблицю експорту DLL, що дозволить використовувати як ім'я функції, так і номер у разі пізнього зв'язування. Зв'язування за номером відбувається швидше.

Додаткове ключове слово NONAME дозволяє експортувати тільки порядковий номер і скоротити завдяки цьому розмір таблиці експорту DLL. Додаткове ключове слово PRIVATE не допускає присутності entryname у бібліотеці імпорту, тобто entryname буде неекспортоване і недоступне для додатків та інших модулів. Додаткове ключове слово DATA указує на те, що експортуються дані, а не код.

Оператори, ключові слова атрибутів і визначені розробником ідентифікатори *є чутливими до регістру символів*. Оператори NAME і LIBRARY повинні передувати будь-яким іншим операторам. Оператори EXPORTS можуть використовуватися в def-файлі більше, ніж один раз. Коментарі в def-файлі позначаються крапкою з комою на початку кожного рядка. Коментар не може перебувати в тому самому рядку, що й оператор, однак він може перебувати між

специфікаціями в багаторядкових операторах. (EXPORTS є багаторядковим оператором). Числові аргументи вказуються в десятковій або шістнадцятковій системі числення.

Трансляція вихідного тексту майбутньої бібліотеки виконується звичайним способом – ML64.EXE /c /coff /Cp /nologo ім'я.asm

Подальше отримання програмного файлу DLL виконується за допомогою спеціального параметра командного рядка лінкера /DLL. Для використання файлу визначення модуля він зазначається за допомогою параметру командного рядка /DEF:ім'я.def.

LINK.EXE /SUBSYSTEM:WINDOWS /RELEASE /DLL /DEF:ім'я.def ім'я.obj

У результаті компонування створюються *два файли*: програмний файл ім'я.dll і файл бібліотеки імпорту ім'я.lib, який використовується для раннього зв'язування додатка з DLL.

Приклад 5.4 Динамічна бібліотека dlldemo.dll абсолютно ідентична за своїми функціональними можливостями раніше розглянутій статичній бібліотеці ssldemo.lib (Приклад 5.**1**).

Статична бібліотека перетворюється на динамічну додаванням у код функції DllMain, заміною атрибута видимості PUBLIC на EXPORT і зміни параметрів командного рядка компонувальника. Наведемо каркасний код DLL (код реалізації функцій такий самий, що і в ssldemo.asm) і продемонструємо використання def-файлу.

```
; Файл dlldemo.asm
.code
DllMain      proc
        mov rax, TRUE
        ret
DllMain endp
StringLength        proc PRIVATE
        .....
```

```
StringLength endp
option prologue:none
option epilogue:none
astrlen proc EXPORT ;size_t strlen(const char* String)
        .....
astrlen endp
amemcpy proc EXPORT ;void *memcpy(void *dst, const void *src, size_t n)
        .....
amemcpy endp
astrcpy     proc EXPORT ;char *strcpy(char *strDestination, const char *strSource)
        .....
astrcpy endp
amemset     proc EXPORT ;void *memset(void *dest, unsigned char c, size_t count)
        .....
amemset     endp
cat2str     proc PRIVATE
        .....
cat2str endp
astrcat     proc EXPORT ; astrcat(unsigned int strcount, ...)
        .....
astrcat endp
end
; Файл dlldemo.def
LIBRARY   dlldemo
EXPORTS
; Функція StringLength не експортується
StringLength                PRIVATE
; Функція astrlen не представлена у def-файлі. Вона експортується тільки
; за іменем за допомогою атрибута видимості EXPORT в описі функції
```

```
; Функції astrcpy, amemcpy, amemset і astrcat експортується за власними
; іменами та ординалами 101, 102, 103 та 104 відповідно
astrcpy         @101
amemcpy         @102
amemset         @103
astrcat         @104
; Функції astrcpy і astrcat також експортується за іншими іменами
; StrCopy і StrCatEx та іншими ординалами 105 і 106
StrCopy     = astrcpy     @105
StrCatEx    = astrcat     @106
```

5.5.4 Зв’язування MASM з динамічними бібліотеками

5.5.4.1 Раннє зв’язування

Зовнішньо раннє зв’язування проекту Microsoft Macro Assembler з динамічною бібліотекою нічим не відрізняється від зв’язування з статичною бібліотекою (розділ 5.4.2). Для раннього зв’язування програмний файл DLL (із розширенням dll) не потрібен і використовується тільки файл бібліотеки імпорту з розширенням lib, який автоматично створюється в результаті компонування DLL. Під час лінкування проєкту файл бібліотеки імпорту повинен розміщуватися в каталозі проекту, або у стандартних каталогах, у яких лінкер виконує пошук бібліотек імпорту, або зазначатися за повним ім’ям.

На момент старту додатка dll-файл повинен розміщуватися у стандартних каталогах, у яких завантажувач виконує пошук DLL.

Додаток на асемблері, що демонструє використання динамічної бібліотеки dlldemo.dll (Приклад 5.4) унаслідок раннього зв’язування можна отримати на підставі вихідного коду файла slldemoasmcaller.asm (приклад 5.5) заміною усього одного рядка includelib slldemo.lib на includelib dlldemo.lib. Тобто MASM використовує одну директиву INCLUDELIB як для зв’язування зі статичними бібліотеками, так і для раннього зв’язування з динамічними бібліотеками.

Потрібно враховувати, що slldemo.lib містить машиний код, а dlldemo.lib тільки інформацію про dlldemo.dll.

5.5.4.2 Пізнє зв'язування

У разі пізнього зв'язування завантаження DLL і визначення адреси потрібної функції виконується самим додатком у процесі його роботи за допомогою функцій Win API LoadLibrary (або LoadLibraryEx) і GetProcAddress.

Функція HMODULE WINAPI LoadLibrary(__in LPCTSTR lpFileName) відображає модуль з ім'ям (рядок символів, що завершується нулем) за адресою lpFileName в адресний простір поточного процесу. Ім'я може зазначатися із заданням шляху і без розширення файлу, яке може бути .dll або .exe.

Якщо рядок визначає відносний шлях або назву модуля без шляху, функція використовує стандартні шляхи для пошуку модуля. Якщо рядок визначає ім'я модуля без шляху, а розширення імені файлу пропущено, функція додає до імені модуля стандартне розширення .DLL.

У разі успіху функція повертає ідентифікатор модуля, інакше NULL. Ідентифікатор модуля в подальшому використовується як один з параметрів функції GetProcAddress.

Функція FARPROC WINAPI GetProcAddress(__in HMODULE hModule, __in LPCSTR lpProcName) у разі успіху повертає адресу функції або змінної з іменем (рядок символів, що завершується нулем) за адресою lpProcName або номером (ординалом) зазначеним у параметрі lpProcName у модулі з ідентифікатором, зазначеним у параметрі hModule. У разі помилки функція повертає NULL.

Функції LoadLibrary (або LoadLibraryEx) і GetProcAddress перебувають у бібліотеці kernel32.dll, із якою попередньо потрібно виконати статичне (раннє) зв'язування.

Приклад 5.5. Додаток на асемблері dlldemodynamiclinkasm.exe, що демонструє використання динамічної бібліотеки dlldemo.dll (Приклад 5.4) з технологією пізнього зв’язування.

```
; Файл dlldemodynamiclinkasm.asm
include \masm64\include64\masm64rt.inc
.data
str1        CHAR "First string",0
str2        CHAR "+second string",0
str3        CHAR "+third string",0
str4        CHAR "+fourth string",0
str5        CHAR "+fifth string",0
str6        CHAR "+sixth string",0
ordinal     CHAR "@101"
.data?
buffer      db 256 dup (?)
dll         HANDLE ?
strlen      PVOID ?
strcpy      PVOID ?    ;@101
memcpy      PVOID ?    ;@102
memset      PVOID ?    ;@103
strcat      PVOID ?    ;@104
.code
main        proc
        call AllocConsole
        mov rcx,cfm$("dlldemo.dll")
        call LoadLibrary
        or rax,rax
        jnz @F
        mov rcx,cfm$("\nUnable to load library")
```

```
        call StdOut
        jmp exit
@@:     mov dll,rax
        mov rcx,rax
        mov rdx,cfm$("astrlen")
        mov rbx,rdx
        call GetProcAddress
        or rax,rax
        jz err
        mov strlen,rax
        mov rcx,dll
        mov rdx,cfm$("astrcpy")
        mov rbx,rdx
        call GetProcAddress
        or rax,rax
        jz err
        mov strcpy,rax
        mov rcx,dll
        mov rdx,cfm$("amemcpy")
        mov rbx,rdx
        call GetProcAddress
        or rax,rax
        jz err
        mov memcpy,rax
        mov rcx,dll
        mov rdx,cfm$("amemset")
        mov rbx,rdx
        call GetProcAddress
        or rax,rax
```

```
jz err
mov memset,rax
mov rcx,dll
mov rdx,cfm$("astrcat")
mov rbx,rdx
call GetProcAddress
or rax,rax
jz err
mov strcat,rax
mov rcx,cfm$("---------- By function name ----------")
call StdOut
lea rsi,str1
lea rdi,buffer
mov rcx,rdi
mov rdx,rsi
call strcpy
mov rcx,cfm$("\nastrcpy:\t")
call StdOut
lea rcx,buffer
call StdOut
sub rsp,(1+4+3)*8 ; return address+shadow+tree parameters
mov rcx,6
lea rdx,buffer
lea r8,str2
lea r9,str3
lea rax,str4
mov [rsp+4*8],rax
lea rax,str5
mov [rsp+5*8],rax
```

```
lea rax,str6
mov [rsp+6*8],rax
call strcat
add rsp,(1+4+3)*8
mov rcx,cfm$("\nstrcat:\t\t")
call StdOut
mov rcx,rdi
call StdOut
mov rcx,rdi
mov rdx,"x"
mov r8,15
call memset
mov rcx,cfm$("\namemset:\t")
call StdOut
mov rcx,rdi
call StdOut
mov rcx,rsi
call strlen
mov rcx,rdi
mov rdx,rsi
mov r8,rax
call memcpy
mov rcx,cfm$("\namemcpy:\t")
call StdOut
mov rcx,rdi
call StdOut
mov rcx,rsi
call strlen
mov rbx,rax
```

```
        rcall vc_printf,cfm$("%s%s%s"),cfm$("\nastrlen:\t\tLength of string \l"),rsi
        rcall vc_printf,cfm$("%s%d"),cfm$("\r equal "),rbx
        lea rdi,strcpy
        mov r12,101
next:   mov rcx,dll
        mov rdx,r12
        call GetProcAddress
        or rax,rax
        jnz @F
        inc byte ptr ordinal+3
        lea rbx,ordinal
        jmp err
@@:     mov qword ptr [rdi],rax
        lea rdi,[rdi+sizeof(PVOID)]
        inc r12
        cmp r12,105
        jb next
        mov rcx,cfm$("\n---------- By function ordinal ----------")
        call StdOut
        lea rsi,str1
        lea rdi,buffer
        mov rcx,rdi
        mov rdx,rsi
        call strcpy
        mov rcx,cfm$("\nastrcpy:\t")
        call StdOut
        lea rcx,buffer
        call StdOut
        sub rsp,(1+4+3)*8 ; return address+shadow+tree parameters
```

```
mov rcx,6
lea rdx,buffer
lea r8,str2
lea r9,str3
lea rax,str4
mov [rsp+4*8],rax
lea rax,str5
mov [rsp+5*8],rax
lea rax,str6
mov [rsp+6*8],rax
call strcat
add rsp,(1+4+3)*8
mov rcx,cfm$("\nstrcat:\t\t")
call StdOut
mov rcx,rdi
call StdOut
mov rcx,rdi
mov rdx,"x"
mov r8,15
call memset
mov rcx,cfm$("\namemset:\t")
call StdOut
mov rcx,rdi
call StdOut
mov rcx,rsi
call strlen
mov rcx,rdi
mov rdx,rsi
mov r8,rax
```

```
call memcpy
mov rcx,cfm$("\namemcpy:\t")
call StdOut
mov rcx,rdi
call StdOut
mov rcx,rsi
call strlen
mov rbx,rax
rcall vc_printf,cfm$("%s%s%s"),cfm$("\nastrlen:\t\tLength of string \l"),rsi
rcall vc_printf,cfm$("%s%d"),cfm$("\r equal "),rbx
mov rcx,cfm$("\n---------- By exported function name ----------")
call StdOut
lea rsi,str1
lea rdi,buffer
mov rcx,rdi
xor rdx,rdx
mov r8,sizeof buffer
call memset
mov rcx,dll
mov rdx,cfm$("StrCopy")
mov rbx,rdx
call GetProcAddress
or rax,rax
jz err
lea rcx,buffer
lea rdx,str1
call rax
mov rcx,cfm$("\nStrCopy:\t")
call StdOut
```

```
        lea rcx,buffer
        call StdOut
        mov rcx,dll
        mov rdx,cfm$("StrCatEx")
        mov rbx,rdx
        call GetProcAddress
        or rax,rax
        jz err
        sub rsp,(1+4+3)*8 ; return address+shadow+tree parameters
        mov rcx,6
        lea rdx,buffer
        lea r8,str2
        lea r9,str3
        lea rax,str4
        mov [rsp+4*8],rax
        lea rax,str5
        mov [rsp+5*8],rax
        lea rax,str6
        mov [rsp+6*8],rax
        call strcat
        add rsp,(1+4+3)*8
        mov rcx,cfm$("\nStrCatEx:\t")
        call StdOut
        mov rcx,rdi
        call StdOut
        jmp exit
err:    mov rcx,cfm$("\nUnable to get function address ")
        call StdOut
        mov rcx,rbx
```

```
        call StdOut
        mov rcx,dll
        call FreeLibrary
exit:   waitkey cfm$("\nPress \lEnter\r ...")
        ret
main endp
end
```

5.5.5 Зв’язування Microsoft Visual C++ із динамічними бібліотеками

5.5.5.1 Раннє зв’язування

Раннє зв’язування проекту Microsoft Visual C++ з динамічною бібліотекою не відрізняється від зв’язування зі статичною бібліотекою (розділ 5.4.3).

Для раннього зв’язування програмний файл DLL (із розширенням dll) непотрібен, і використовується тільки файл бібліотеки імпорту з розширенням lib, який автоматично створюється в результаті компонування DLL.

Додаток на C++ (Microsoft Visual C++), що демонструє використання динамічної бібліотеки dlldemo.dll (Приклад 5.4) за технологією раннього зв’язування можна отримати на підставі вихідного коду файла slldemoasmcaller.asm (приклад 5.6) заміною імені файлу бібліотеки в командному рядку лінкера в настройках властивостей проєкту (або скористатися директивою #pragma comment(linker, "/DEFAULTLIB:ім’я.lib")) з slldemo.lib на dlldemo.lib.

5.5.5.2 Пізнє зв’язування

Пізнє зв’язування з DLL на C++ вимагає попереднього визначення типів для кожної функції з подальшим приведенням результату, що повертає GetProcAddress на цей тип.

Приклад 5.6. Додаток dlldemodynamiclinkcpp.exe, що демонструє використання динамічної бібліотеки dlldemo.dll (Приклад 5.4) за технологією пізнього зв’язування.

```
; Файл dlldemodynamiclinkcpp.cpp
#include <windows.h>
#include <iostream>
using namespace std;
typedef DWORD(*dllstrlen)(LPSTR);
typedef void(*dllmemcpy)(LPSTR lpdst, LPSTR lpsrc, DWORD n);
typedef void(*dllstrcpy)(char*, char*);
typedef void(*dllfcstrcpy)(char* lpdst, char* lpsrc);
typedef void(*dllmemset)(char* lpdest, char chr, unsigned int count);
typedef void(*dllfcmemset)(char* lpdest, char chr, unsigned int count);
typedef void(*dllstrcat)(unsigned int strcount, ...);
int main()
{
	char str1[] = "First string";
	char str2[] = "+second string";
	char str3[] = "+third string";
	char str4[] = "+fourth string";
	char str5[] = "+fifth string";
	char str6[] = "+sixth string";
	char buffer[256];
	HMODULE		dll;
	dllmemcpy	pmemcpy;
	dllstrcpy	pstrcpy;
	dllmemset	pmemset;
	dllstrcat	pstrcat;
	dllstrlen	pstrlen;
	if ((dll = LoadLibrary(L"dlldemo.dll")) != NULL)
	{
		cout << "---------- By original name ----------" << endl;
```

```
        //astrcpy(buffer,str1);
        pstrcpy = (dllstrcpy)GetProcAddress(dll, "astrcpy");
        pstrcpy(buffer, str1);
        cout << "astrcpy:\t" << buffer << endl;
        //astrcat(6,buffer, str2, str3, str4, str5, str6);
        pstrcat = (dllstrcat)GetProcAddress(dll, "astrcat");
        pstrcat(6, buffer, str2, str3, str4, str5, str6);
        cout << "astrcat:\t" << buffer << endl;
        //amemset(buffer,'x',15);
        pmemset = (dllmemset)GetProcAddress(dll, "amemset");
        pmemset(buffer, 'x', 15);
        cout << "amemset:\t" << buffer << endl;
        //amemcpy(buffer,str1,astrlen(str1)+1);
        pmemcpy = (dllmemcpy)GetProcAddress(dll, "amemcpy");
        pstrlen = (dllstrlen)GetProcAddress(dll, "astrlen");
        pmemcpy(buffer, str1, pstrlen(str1));
        cout << "amemcpy:\t" << buffer << endl;
        cout << "astrlen:\tLength of string '" << str1 << "' equal " << pstrlen(str1)
<< endl;
        cout << "---------------------- By ordinal ----------------------------" << endl;
        //amemset(buffer,0,astrlen(str1));
        pmemset(buffer, 0, pstrlen(str1));
        //astrcpy(buffer,str1);
        pstrcpy = (dllstrcpy)GetProcAddress(dll,
                        (LPCSTR)MAKEINTRESOURCE(101));
        pstrcpy(buffer, str1);
        cout << "astrcpy:\t" << buffer << endl;
        //astrcat(6,buffer, str2, str3, str4, str5, str6);
        pstrcat = (dllstrcat)GetProcAddress(dll,
```

```
                                (LPCSTR)MAKEINTRESOURCE(104));
            pstrcat(6, buffer, str2, str3, str4, str5, str6);
            cout << "astrcat:\t" << buffer << endl;
            //amemset(buffer,'x',15);
            pmemset = (dllmemset)GetProcAddress(dll,
                                        (LPCSTR)MAKEINTRESOURCE(103));
            pmemset(buffer, 'x', 15);
            cout << "amemset:\t" << buffer << endl;
            //amemcpy(buffer,str1,astrlen(str1)+1);
            pmemcpy = (dllmemcpy)GetProcAddress(dll,
                                         (LPCSTR)MAKEINTRESOURCE(102));
            pmemcpy(buffer, str1, pstrlen(str1));
            cout << "amemcpy:\t" << buffer << endl;
            cout << "astrlen:\tLength of string '" << str1 << "' equal " << pstrlen(str1)
<< endl;
            cout << "---------------------- By exported name ---------------------" << endl;
            //amemset(buffer,0,astrlen(str1));
            pmemset(buffer, 0, pstrlen(str1));
            pstrcpy = (dllstrcpy)GetProcAddress(dll, "StrCopy");
            //StrCopy(buffer,str1);
            pstrcpy(buffer, str1);
            cout << "StrCopy:\t" << buffer << endl;
            pstrcat = (dllstrcat)GetProcAddress(dll, "StrCatEx");
            //StrCatEx(6,buffer, str2, str3, str4, str5, str6);
            pstrcat(6, buffer, str2, str3, str4, str5, str6);
            cout << "StrCatEx:\t" << buffer << endl;
        }
        else
        {
```

```
            MessageBox(0, L"Не можу завантажити бібліотеку", NULL,
                                    MB_OK | MB_ICONERROR);
            return -1;
        }
        return 0;
}
```

6 МАКРОЗАСОБИ MICROSOFT MACRO ASSEMBLER

Макрозасоби надають можливість розробляти гарно стилізований і структурований код. Використання макрозасобів дозволяє спростити і скоротити вихідний текст програми, зробивши його більш зрозумілим і зменшивши кількість можливих помилок кодування. Макрозасоби являють собою потужний мовний засіб асемблерів і роблять програмування на асемблері більш наочним, легким, витонченішим і схожим на програмування мовами високого рівня.

Макрозасоби – це певний набір сервісних послуг асемблера, який реалізується програмними засобами транслятора. Для асемблерів, які підтримують макрозасоби, транслятор складається з двох частин – препроцесора (макроасемблера) і власне транслятора (асемблера).

Вихідний текст програми, що використовує макрозасоби, може бути взагалі не схожим на асемблерну програму. Спочатку цей вихідний текст обробляться препроцесором, що виконує так звану макрогенерацію, готуючи технологічний лістинг для подальшої обробки транслятором. У процесі макрогенерації виконується «переклад» з мови препроцесора мовою, зрозумілою для транслятора.

Набір макрозасобів (мова препроцесора) різниться для різних пакетів асемблерів, хоча існує певна їх частина, яка підтримується більшістю пакетів у незмінному синтаксисі. Певно найбільшу кількість власних директив і операторів має препроцесор FASM, однак не дуже поступається йому і MASM,

повна назва якого Microsoft Macro Assembler наголошує на його орієнтованості на використання макрозасобів.

Макрозасоби MASM можна розділити на кілька основних груп

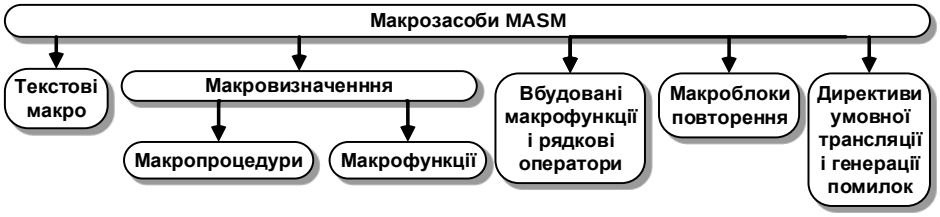


Рисунок 6.1 – Групи макрозасобів MASM

Кожна наступна група макрозасобів, зображених на рисунку зліва направо, розширюватиме можливості щодо макропрограмування.

6.1 Текстові макро

Текстовий макро дозволяє призначити ім’я (скорочення, аліас) певній послідовності символів, а потім використати це ім’я замість тексту у вихідному коді. Визначається текстовий макро за допомогою директиви TEXTEQU в одній із таких форм:

name TEXTEQU <Текст>

name TEXTEQU %Константний вираз

name TEXTEQU Інший текстовий макро

name TEXTEQU Макрофункція

Символи «<», «>», «%» є макрооператорами препроцесора. Лапки з символів < > визначають літерали, указуючи препроцесору, що він повинен сприймати щось укладене у них як єдиний рядок символів, навіть якщо він містить коми, пробіли та будь-які інші символи. При цьому самі символи «<» і «>» у рядок не потрапляють.

Якщо права частина текстового макро містить символи «<», «>», «,» «"», «'», «%», «;» які можуть мати певну функціональність, слід екранувати (затінити, відмінити) за допомогою макрооператора «!», який повідомляє транслятору про те, що наступний за ним символ слід уважати просто символом, навіть якщо це кома, кутова дужка або щось інше.

У константних арифметичних виразах можуть використовуватися оператори LENGTH, SIZE, WIDTH, MASK, LENGTHOF, SIZEOF, унарні «+» і «-», знаки бінарних операцій «+»,«-», «*», «/», MOD, оператори побітового зрушення SHL, SHR, логічні побітові операції NOT, AND, OR, XOR, дужки «(» та «)».

Операндами арифметичних виразів можуть бути числа, попередньо визначені макрозмінні, макропараметри. У арифметичних виразах не повинно бути нічого, що препроцесор не може обчислити *на етапі трансляції*. Препроцесор MASM не розрізняє знакові та беззнакові двійкові і шістнадцяткові числа, значення числа не повинно виходити за діапазон подвійного слова, препроцесор ніяк не контролює переповнення.

Макрооперанд «%» використовується для того, щоб повідомити препроцесору про те, що весь наступний за ним текст являє собою константний вираз або текстовий макро, який потрібно *попередньо обчислити* або розгорнути, і надалі використовувати результат *у текстовому вигляді*.

6.2 Макровизначення

Використання текстових макро певним чином підвищує наочність вихідного тексту програми, але за допомогою них у програмі можна замінити не більше одного рядка. Подальшим розвитком механізму макропідстановок і макрогенерації є використання макровизначень.

За допомогою макровизначень до вихідного тексту програми можна вставляти цілі послідовності машинних команд, директив визначення даних,

коментарів і будь-яких інших синтаксичних конструкцій асемблера, що можуть бути у програмі. Найпростіше макровизначення має такий вигляд:

name MACRO

Макровизначення

ENDM

Макровизначення розміщують на початку вихідного тексту програми або в окремому файлі. Для розміщення тексту макровизначення в окремому файлі на початку програми необхідно задати директиву INCLUDE [[диск:\]шлях\]ім'я файлу. Зустрівши директиву INCLUDE, препроцесор знайде зазначений файл і вставить *увесь його текст* до технологічного лістингу програми, після чого продовжить подальше оброблення. Отже, директиву INCLUDE можна вважати певним розширенням текстових макро. За допомогою INCLUDE до програми можна вставляти текст будь-якого файлу, і можна розробити таку програму, вихідний текст якої міститиме лише директиви INCLUDE.

У тексті макровизначення можна використовувати директиву EXITM, і за наявністю у макровизначенні цієї директиви препроцесор MASM розрізняє *макропроцедури та макрофункції*. Макровизначення які містять директиву EXITM, *можуть повертати* певне значення і вважаються макрофункціями.

Директива EXITM схожа на оператор RETURN мови C/C++ і має такий синтаксис: EXITM [ReturnValue]. Коли препроцесор зустрічає директиву EXITM у макровизначенні, подальше виконання макро припиняється, і повертається *необов'язковий* параметр ReturnValue.

Значенням ReturnValue, що повертається макрофункцією за допомогою директиви EXITM *повинен бути текст* і макрофункції повинні за потреби спочатку перетворити числове значення на текст перед його поверненням. За відсутності значення ReturnValue в директиві EXITM повертається *порожній рядок тексту*.

За зовнішніми ознаками використання макропроцедур і макрофункцій в асемблері схоже на використання процедур і функцій мов високого рівня і, ураховуючи таку їхню схожість, використовують визначення «виклик макропроцедури» або «виклик макрофункції» (у C/C++, наприклад, неможливо на рівні вихідного тексту відрізнити макропроцедуру чи макрофункцію від справжньої функції). Також використовуватимемо термін «виклик», однак при цьому слід пам'ятати, що відносно функції «виклик» означає команду передавання керування, а для макропроцедури чи макрофункції – *вставку у вихідний текст у місце «виклику» тексту макровизначення*.

Макропроцедури і макрофункції розрізняються за способом виклику. Макропроцедура викликається зазначенням імені відповідного макровизначення з початку нового рядка програми. Макрофункція викликається зазначенням імені відповідного макровизначення, за яким обов'язково повинні бути присутні символи «(» та «)». Макрофункція може бути частиною будь-яких припустимих виразів, у тому числі інших макрофункцій.

Використання порожніх дужок «()» у запису виклику макрофункції синтаксично нагадує виклик функцій C/C++, які не мають параметрів. Це не лише зовнішня схожість – макропроцедури та макрофункції дійсно можуть мати формальні параметри, які на етапі макрогенерації будуть замінюватися фактичними параметрами. Отже, повний синтаксис макровизначення має такий вигляд:

```
name MACRO [Параметри]
        Макровизначення
        [EXITM [ReturnValue]]
ENDM
```

Параметри макровизначення (макропараметри) являють собою розділений комами перелік символьних імен *формальних параметрів*. Під час виклику

макропроцедури отримують розділений комами перелік *фактичних параметрів* через пробіл після свого імені, а макрофункції – у дужках «()» після свого імені. Для перенесення параметрів на наступний рядок, як для опису макровизначення, так і для його виклику, використовується символ «\» після коми.

Якщо фактичний параметр макровизначення містить символ коми, який може сприйматися препроцесором як символ відокремлення одного параметра від іншого, то в цьому випадку використовують оператор визначення літерала «<>».

Перелік макропараметрів може складатися з елементів із синтаксисом чотирьох типів:

$$\text{Елемент переліку}=\begin{cases}\text{макропараметр}\\ \text{макропараметр:=<значення>}\\ \text{макропараметр:REQ}\\ \text{макропараметр:VARARG}\end{cases}$$

Кількість фактичних параметрів для виклику макровизначення може бути меншою, ніж кількість формальних параметрів, зазначена в описі макровизначення (фактичні параметри можуть бути взагалі відсутні). Під час виклику макровизначення з меншою кількістю фактичних параметрів для збереження позицій параметрів залишають символи коми.

Наприклад, визначимо макропроцедуру, яка за допомогою оператора ECHO (або %OUT) виводить у стандартний потік виведення препроцесора (на екран) значення своїх параметрів

```
MacroNameMACRO a,b,c
        echo a b c
ENDM
```

Виклик макропроцедури може виконуватися так (справа за символом коментаря зазначено виведення ECHO):

```
MacroName 1,2,3          ; 1 2 3
```

```
MacroName 1,,3          ; 1  3
MacroName ,2,3          ; 2 3
MacroName 1,2           ; 1 2
MacroName 1             ; 1
MacroName               ;
```

За допомогою оператора «:=» параметру можна привласнити значення за замовчуванням, яке він буде приймати, якщо відповідний фактичний параметр не зазначено під час виклику макровизначення. Якщо попередню макропроцедуру визначити так:

```
MacroName       MACRO     a:=<4>,b:=<5>,c:=<6>
        echo a b c
ENDM,
```

то її можна викликати як і в попередньому прикладі, але результати її роботи (коментар справа) тепер відрізнятимуться:

```
MacroName 1,2,3         ; 1 2 3
MacroName 1,,3          ; 1 5 3
MacroName ,2,3          ; 4 2 3
MacroName 1,2           ; 1 2 6
MacroName 1             ; 1 5 6
MacroName               ; 4 5 6
```

Тип параметра REQ вказує препроцесору на те, що цей параметр повинен *обов'язково* зазначатися і не може бути пропущений під час виклику макровизначення. Якщо макропроцедуру визначити так:

```
MacroName       MACRO     a:REQ,b:REQ,c:REQ
        echo a b c
ENDM,
```

то всі раніше розглянуті способи її виклику, окрім MacroName 1, 2, 3 завершуватимуться з помилкою препроцесора.

Інші можливості щодо контролю та обробки фактичних параметрів макровизначення надають директиви умовної трансляції та генерації помилок, які будуть розглянуті пізніше.

Приклад. Наступні дві макропроцедури дозволяють використовувати в програмах на асемблері псевдокоманду movm для пересилання «пам'ять–пам'ять» і псевдокоманду return схожу на однойменний оператор C/C++.

```
movm MACRO    dst:REQ,src:REQ
              push src
              pop dst
ENDM
return MACRO code:=<0>
              mov eax,code
              ret
ENDM
```

Під час "виклику" макровизначення препроцесор виконує макрогенерацію, замінюючи спочатку формальні макропараметри *текстом* зазначених при виклику фактичних параметрів, а потім розгортає макровизначення. Розробляючи макровизначення, слід розуміти, що макропараметр – це *замінник тексту* майбутнього фактичного параметра і його значення може бути використане безпосередньо або призначене іншому символу, але *не може бути змінене*. Водночас макровизначення може інтерпретувати *текст* фактичного параметра, який воно отримує, або як числове значення, або як текстове значення.

Посилання на макропараметр у межах макровизначення можуть іноді бути неоднозначні, і препроцесор може їх не розпізнати і не розширити. Оператор макропідстановки & дозволяє однозначно ідентифікувати будь-який параметр у межах макровизначення.

Використання («виклик») макровизначення призводить до вставляння у відповідне місце програми *тексту* макровизначення. За допомогою

макровизначень до вихідного тексту програми можна вставляти будь-які синтаксичні конструкції асемблера, зокрема послідовності машинних команд із символічними іменами міток, імена змінних програми з директивами визначення даних.

Виклик одного і того самого макровизначення два і більше рази може призвести до того, що у вихідному тексті програми з'являться однакові символічні імена міток, змінних тощо. Така ситуація не контролюється препроцесором і є для нього абсолютно нормальним результатом макрогенерації. Однак після препроцесора відпрацьовує транслятор, який справедливо вважає, що програма не повинна мати декілька однакових міток, процедур, змінних.

Розв'язати подібну проблему можна введенням додаткових макропараметрів і використанням оператора макропідстановки &, однак MASM надає для цього зручніший засіб – директиву LOCAL. Ця директива *має бути розміщена безпосередньо за макрозаголовком* і містити перелік розділених комами використовуваних у макровизначенні символічних імен.

Приклад. Наступна макрофункція дозволяє використовувати однобайтові рядки символів, що завершуються нулем безпосередньо в коді без попереднього використання директив визначення даних у секції ініціалізованих даних.

Добавлено примечание ([АЮ1]):

```
TEXT        MACRO message
LOCAL szText
.data
szText BYTE message,0
align 8
.code
EXITM <ADDR szText>
ENDM
.code
.....
```

mov rsi,TEXT("Macro function demo")

Тип параметра VARARG може зазначатися тільки останнім і вказує препроцесору на те, що все, починаючи з коми, що передує цьому параметру, слід *розглядати як єдиний рядок символів*, навіть якщо він містить коми. Цей тип параметра потребує окремої обробки в макровизначенні, яку зручно виконувати з використанням вбудованих макрофункцій і рядкових операторів та макроблоків повторення.

6.3 Вбудовані макрофункції та рядкові оператори

Для обробки тексту препроцесор MASM надає вбудовані макрофункції та рядкові оператори, які за функціональним призначенням можна розділити на чотири групи.

Макрофункція @SizeStr(string) повертає кількість символів у рядку string.

Макрофункція @InStr([start], string, substring) повертає позицію (номер символу) у рядку string починаючи з якої починається підрядок substring. Пошук починається з позиції start, або від початку рядка (позиції 1), якщо start не зазначений. Якщо @InStr не знаходить підрядок – повертається значення 0.

Макрофункція @CatStr(string1[,string2]...) повертає рядок, отриманий конкатенацією рядків string1, string2 і т. д.

Макрофункція @SubStr(string, start[,length]) повертає підрядок від рядка string, починаючи з позиції start довжиною length символів. Якщо length відсутній, @SubStr повертає залишок рядка, починаючи з позиції start.

Слід пам’ятати про те, що *всі макрофункції повертають текст* (@SizeStr і @InStr повертають символьне подання числа). Поряд з розглянутими макрофункціями існують рядкові оператори SIZESTR, INSTR, CATSTR, SUBSTR, які мають синтаксис:

name SIZESTR string

name INSTR [start,] string, substring

name CATSTR string1[, string2]...

name SUBSTR string, start[,length]

Рядкові оператори за способом використання схожі на текстові макро, а за функціональністю аналогічні відповідним макрофункціям. Вищезазначені чотири рядкові оператори еквівалентні, відповідно:

name TEXTEQU @SizeStr(string)

name TEXTEQU @InStr([start], string, substring)

name TEXTEQU @CatStr(string1[, string2]...)

name TEXTEQU @SubStr(string, start[,length])

6.4 Макроблоки повторення

У макроблоці повторення

FOR параметр[:REQ|:=значення], <перелік параметрів>

Макровизначення

ENDM

параметр по черзі приймає значення кожного з переліку *розділених комою* параметрів.

У макроблоці повторення

FORC параметр,<рядок>

Макровизначення

ENDM

параметр по черзі приймає значення кожного із символів рядка

Макроблок

REPEAT константний вираз

Макровизначення

ENDM

повторює макровизначення кількість разів, що визначається значенням константного виразу.

Макроблок

WHILE логічний вираз

Макровизначення

ENDM

повторює макровизначення, доки вираз набуває значення «істина». У логічних виразах можна використовувати оператори співвідношення (Таблиця 6.1) і логічні оператори (таблиця 6.2)

Таблиця 6.1 – Оператори співвідношення

Синтаксис	Результат
вираз1 EQ вираз2	Істина, якщо вираз1 дорівнює вираз2
вираз1 NE вираз2	Істина, якщо вираз1 не дорівнює вираз2
вираз1 LT вираз2	Істина, якщо вираз1 менше вираз2
вираз1 LE вираз2	Істина, якщо вираз1 менше або дорівнює вираз2
вираз1 GT вираз2	Істина, якщо вираз1 більше вираз2
вираз1 GE вираз2	Істина, якщо вираз1 більше або дорівнює вираз2

Таблиця 6.2 – Логічні оператори

Синтаксис	Результат
NOT вираз	Істина, якщо вираз брехливий, і навпаки
вираз1 AND вираз2	Істина, якщо вираз1 і вираз2 обидва істинні
вираз1 OR вираз2	Істина, якщо вираз1 або вираз2 істина
вираз1 XOR вираз2	Істина, якщо вираз1 дорівнює запереченню вираз2

Приклад. Обчислення факторіала засобами препроцесора.

```
factorial   MACRO   number:REQ
  LOCAL   i, f
```

```
    f = number
    i = 1
    WHILE f GT 1
      i = i * f
      f = f - 1
    ENDM
    EXITM %i
ENDM
.data
n       DWORD factorial(5)
```

6.5 Директиви умовної трансляції і генерації помилок

Директиви умовної трансляції мають загальний синтаксис:

```
IFxxx ключ
        Макровизначення
[ELSEIFxxx ключ]
        Макровизначення
[ELSEIFxxx ключ]
        Макровизначення
……
[ELSE]
        Макровизначення
ENDIF
```

Символи «xxx» визначають різні форми директив умовної трансляції:

1. IF/IFE <логічний вираз> – якщо вираз набуває ненульового/нульового значення. У логічних виразах можна використовувати оператори співвідношення (Таблиця 6.1) і логічні оператори (

Таблиця 6.**2** – Логічні оператори);

2. IFDEF/IFDNDEF <ідентифікатор> – якщо в програмі визначено/не визначено <ідентифікатор> (у тому числі за допомогою EXTRN);

3. IFB/IFNB <рядок> – якщо рядок порожній/непорожній;

4. IFIDN/IFDIF <рядок1>, <рядок2> – якщо рядки однакові/різні;

5. IFIDNI/IFDIFI <рядок1>, <рядок2> – якщо рядки однакові/різні без урахування регістру символів.

MASM має єдину директиву безумовної генерації помилок .ERR, виявивши яку препроцесор завершує обробку. На підставі директиви .ERR з використанням директив умовної трансляції можна організувати доволі складні перевірки правильності функціонування макровизначень. Для простих випадків достатніми виявляються можливості директив умовної генерації помилок, наведених у таблиці 6.3.

Таблиця 6.3 – Директиви умовної генерації помилок

Директива	Причина генерації помилки
.ERRE вираз	якщо вираз дорівнює нулю
.ERRNZ вираз	якщо вираз не дорівнює нулю
.ERRDEF ідентифікатор	якщо ідентифікатор визначено
.ERRNDEF ідентифікатор	якщо ідентифікатор не визначено
.ERRB рядок	якщо рядок порожній
.ERRNB рядок	якщо рядок непорожній
.ERRDIF рядок1, рядок2	якщо рядки різні
.ERRDIFI рядок1, рядок2	якщо рядки різні без урахування регістру символів
.ERRIDN рядок1, рядок2	якщо рядки однакові
.ERRIDNI рядок1, рядок2	якщо рядки однакові без урахування регістру символів

Наприклад, відмінність

```
.ERRNDEF Name
```

від

```
IFNDEF Name
        ECHO &Name not defined
```

```
        .ERR
ENDIF
```

полягає лише в тому, що за допомогою ECHO в IFNDEF можна сформувати розширений коментар, щодо причин завершення роботи препроцесора замість стандартного «error A2055 : forced error : symbol not defined : Name», що генерується .ERRNDEF.

Приклад 6.1. Наступний приклад демонструє використання деяких з вищерозглянутих макрозасобів на прикладі задачі обробки одновимірного масиву. Додаток SumaMacro використовує чотири різні макровизначення для визначення суми елементів одновимірного масиву з елементами різного розміру.

```
; Файл SumaMacro.inc
SumArray1 MACRO     Name,type,reg1,reg2
LOCAL       next
        push rcx
        push rsi
        push rax
        xor rbx,rbx
        mov rcx,LENGTHOF Name
        lea rsi,Name
next:
        lods&type
        add reg1,reg2
        loop next
        pop rax
        pop rsi
        pop rcx
ENDM
SumArray2  MACRO     Name,type,plus
LOCAL       next
```

```
        FOR Register,<rax,rcx,rsi>
                push Register
        ENDM
        xor rbx,rbx
        mov rcx,LENGTHOF Name
        lea rsi,Name
next:   lods&type
        plus
        loop next
        FOR Register,<rsi,rcx,rax>
                pop Register
        ENDM
ENDM
SumArray3 MACRO     Name,sum,type
LOCAL     next
        FORC l,cda
                push r&l&x
        ENDM
        xor sum,sum
        mov rcx,LENGTHOF Name
        lea rdx,Name
next:add sum, type ptr [rdx]
        add rdx, SIZE Name
        loop next
        FORC l,adc
                pop r&l&x
        ENDM
ENDM
pushl MACRO     list:VARARG
```

```
        FOR operand,<list>
                push operand
        ENDM
ENDM
popl    MACRO     list:VARARG
        FOR operand,<list>
                pop operand
        ENDM
ENDM
SumArray4 MACRO      Name
LOCAL       next
        IFB <Name>
                ECHO Array name required
                .ERR
        ENDIF
        IFNDEF Name
                ECHO &Name not defined
                .ERR
        ENDIF
        pushl rax,rcx,rsi
        xor rbx,rbx
        mov rcx,SIZEOF Name / SIZE Name
        lea rsi,Name
next:
        IF SIZE Name EQ 1
                lodsb
                add bl,al
        ELSEIF SIZE Name EQ 2
                lodsw
```

```
        add bx,ax
    ELSEIF SIZE Name EQ 4
        lodsd
        add ebx,eax
    ELSEIF SIZE Name EQ 8
        lodsq
        add rbx,rax
    ELSE
        .ERR
    ENDIF
    loop next
    popl rsi,rcx,rax
ENDM
; Файл SumaMacro.asm
include SumaMacro.inc
.data
ByteArray       db 1,2,3,4,5
WordArray       dw 1,2,3,4,5
DWordArray      dd 1,2,3,4,5
QWordArray      dq 1,2,3,4,5
.code
main        proc
    SumArray1 ByteArray,b,bl,al
    SumArray1 WordArray,w,bx,ax
    SumArray1 DWordArray,d,ebx,eax
    SumArray1 QWordArray,q,rbx,rax

    SumArray2 ByteArray,b,<add bl,al>
    SumArray2 WordArray,w,<add bx,ax>
```

```
        SumArray2 DWordArray,d,<add ebx,eax>
        SumArray2 QWordArray,q,<add rbx,rax>

        SumArray3 ByteArray,bl,byte
        SumArray3 WordArray,bx,word
        SumArray3 DWordArray,ebx,dword
        SumArray3 QWordArray,rbx,qword

        SumArray4 ByteArray
        SumArray4 WordArray
        SumArray4 DWordArray
        SumArray4 QWordArray
        ret
main endp
end
```

7 ПРОГРАМУВАННЯ МАТЕМАТИЧНОГО СПІВПРОЦЕСОРА

Для виконання операції з плаваючою комою у процесорах Intel64/AMD64 використовується спеціальний пристрій – Floating Point Unit (FPU). Це спеціалізований процесор із власними регістрами і власним набором команд, який спочатку постачався у вигляді окремої мікросхеми (i8087, i80287, i80387), що отримала назву математичного співпроцесора і уставлялася в спеціально передбачене для неї рознімання на системній платі, а починаючи з i80486DX4-100 вбудовується в мікросхему основного (CPU) процесора.

Не зважаючи на фізичне поєднання основного процесора і математичного співпроцесора в одному корпусі інтегральної мікросхеми, *логічно* це два різні обчислювальні пристрої. Процесор і математичний співпроцесор можуть працювати паралельно, *виконуючи різні команди*, однак повної паралельності в

їх роботі немає, оскільки процесор забезпечує математичному співпроцесору інтерфейс з пам’яттю. Синхронізація доступу до пам’яті з боку процесора і співпроцесора забезпечується в більшості випадків апаратними засобами і для програмування є абсолютно прозорою.

7.1 Програмна модель математичного співпроцесора

Під програмною моделлю математичного співпроцесора, так само, як і під програмною моделлю основного процесора, розуміють його організацію щодо можливостей програмування. Рисунок 7.1 відображає складові програмної моделі математичного співпроцесора.

Рисунок 7.1 – Програмна модель співпроцесора

Із урахуванням того, що математичний співпроцесор є складовою частиною основного процесора, у контексті його програмної моделі не розглядаються режими роботи, які він «успадковує» від основного процесора, тому вони не подані на рисунку.

Оскільки основний процесор забезпечує математичному співпроцесору інтерфейс з пам’яттю, способи організації та адресації пам’яті для них збігаються. Вони зображені на рисунку тому, що в системі команд співпроцесора є команди, які передбачають використання операндів у пам’яті. Для таких операндів будуть використовуватися відомі вже з попередніх розділів способи адресації пам’яті.

7.1.1 Типи даних математичного співпроцесора

Математичний співпроцесор підтримує такі типи даних.

1. Двійкові цілі (знаковий біт у старшому розряді) числа в трьох форматах:
   а) 16-бітне ціле число від -32 768 до 32 767;
   б) 32-бітне коротке ціле від $-2 \cdot 10^9$ до $+2 \cdot 10^9$;
   в) 64-бітне довге ціле від $-9 \cdot 10^{18}$ до $+9 \cdot 10^{18}$.
2. Числа з плаваючою комою в трьох форматах:
   а) коротке 32-бітне дійсне число від 10^{-38} до 10^{+38};
   б) довге 64-бітне дійсне число від 10^{-308} до 10^{+308};
   в) розширене 80-бітне дійсне число від 10^{-4932} до 10^{+4932}.
3. Упаковані цілі BCD-числа у форматі 9+1 – у дев'яти молодших байтах розміщуються числові розряди, а в десятому байті використовується тільки старший біт для кодування знаку).
4. Спеціальні числові значення:
   а) позитивний і негативний нуль;
   б) позитивна і негативна нескінченність;
   в) денормалізовані дійсні числа;
   г) сигнальні не числа SNAN і тихі не числа QNAN.

Визначення в програмі 16/32/64-бітних двійкових цілих виконується звичним способом за допомогою директив, відповідно, WORD / DWORD / QWORD.

Визначення в програмі 32/64/80-бітних чисел з плаваючою комою може виконуватися за допомогою директив, відповідно, (DD | DWORD | REAL4) / (DQ | QWORD | REAL8) / (DT | REAL10) двома способами – у природній або в експоненціальній формі. *У будь-якій формі в записі числа повинен бути символ «.» десяткової крапки*, навіть якщо число не має дробової частини:

```
.data
X1    DD 123.
Y1    QWORD 123456.0
Z1    DT 123.456
X2    REAL4 1.23e+2
```

Y2 REAL8 –0.123456+e6

Z2 REAL10 123456.0e-3

Визначення в програмі упакованих цілих BCD-чисел виконується звичним способом за допомогою директиви DT. Спеціальні числові значення з'являються у результаті обчислень і, зазвичай, попередньо не визначаються.

7.1.2 Склад і призначення регістрів співпроцесора

У програмній моделі математичного співпроцесора (Рисунок 7.2) можна виділити три групи регістрів.

Основу моделі складає група з восьми 80-розрядних регістрів даних R0–R7 у яких зберігаються числа з плаваючою комою в розширеному форматі: старший (79-й) біт визначає знак числа, у бітах 78–64 розташована так звана характеристика числа, а в бітах 63–0 розміщується його мантиса.

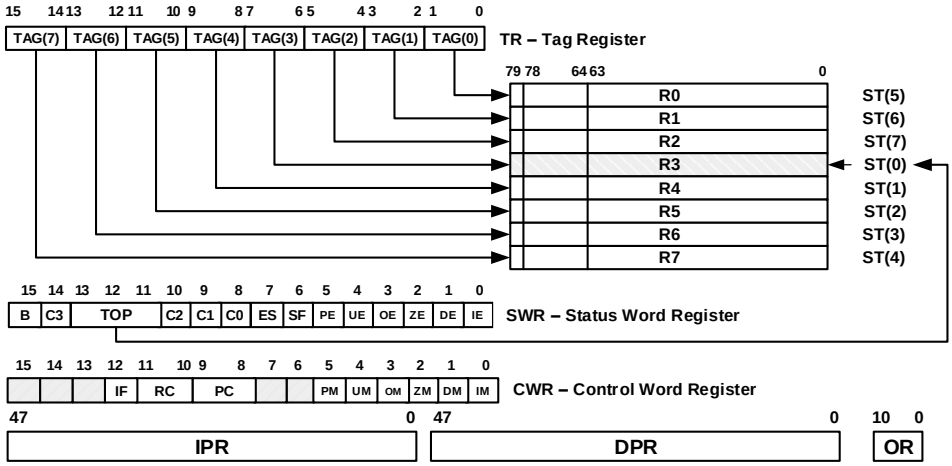


Рисунок 7.2 – Програмна модель математичного співпроцесора

Мантиса зберігається у нормалізованому вигляді, у якому вона завжди починається з одиниці. Під час нормалізації кома в записі числа переміщується вправо або вліво так, щоб в цілій частині числа була одиниця. Кількість

перенесень коми визначає порядок числа s, який зберігається у полі характеристики як значення q=s+16383.

Із завантаженням з пам'яті в регістри R0-R7 даних будь-яких інших типів (7.1.1) вони *автоматично перетворюються на розширений формат* числа з плаваючою комою, і уся їх подальша обробка відбувається в цьому форматі.

Унаслідок вивантаження результатів у пам'ять, якщо вони зберігаються там не як числа з плаваючою комою в розширеному форматі, то автоматично відбувається потрібне перетворення. Із перетворенням у більш короткі формати чисел із плаваючою комою «зайві» розряди після округлення *відкидаються*.

Із кожним *фізичним* регістром R0–R7 жорстко зв'язане (Рисунок 7.2) відповідне двохбітове поле в регістрі тегів співпроцесора (TR, Tag Register). Двохбітове число, що зберігається у відповідних бітах TR, у кожний момент часу надає інформацію про вміст відповідного регістру:

00 – у регістрі знаходиться припустиме ненульове значення;

01 – у регістрі знаходиться нульове значення;

10 – у регістрі знаходиться спеціальне числове значення;

11 – регістр порожній.

Регістри даних математичного співпроцесора організовано у вигляді стека і команди співпроцесора звертаються до регістрів даних R0–R7 за *логічними номерами* ST(0)–ST(7). Логічний номер ST(0) позначає верхівку стека математичного співпроцесора, а ST(1), ST(2), …, ST(7) – елементи вглибині стека.

Верхівка стека ST(0) жорстко не зв'язана ні з яким номером фізичного регістра. Стек співпроцесора організовано у вигляді кільцевого буфера, в якому верхівка стека ST(0) «плаває» і логічний номер ST(0) у будь-який момент часу може вказувати на будь-який фізичний регістр. Поточне положення верхівки стека математичного співпроцесора визначається значенням 3-бітового поля TOP (Рисунок 7.2) регістра слова стану співпроцесора (SWR, Status Word

Register). Положення елементів ST(1)–ST(7) визначається відносно ST(0) із замиканням на кільце.

Команди завантаження даних з пам'яті у стек співпроцесора спочатку автоматично зменшують значення TOP на одиницю, а після цього розміщують дані вже в новій верхівці стека. Команди вивантаження даних спочатку копіюють дані з верхівки стеку співпроцесора в пам'ять, а потім і*ще можуть* (залежить від команди) автоматично збільшувати значення TOP на одиницю.

У цілому Status Word Register виконує функції аналогічні регістру прапорців RFLAGS основного процесора – в будь-який момент часу біти регістра SWR відображають поточний стан математичного співпроцесора загалом, і після виконання ним кожної команди зокрема.

Старший біт B (біт 15) SWR використовується тільки для сумісності з попередніми моделями математичних співпроцесорів – його значення завжди віддзеркалює значення сьомого біту ES.

Біти C3, C2, C1, C0 відображають певні стани співпроцесора та результати виконання команд, однак при цьому їх інтерпретація залежить від ситуації та команди. *У межах системи команд математичного співпроцесора, вплив команд на стан SWR, у більшості випадків не розглядатиметься*. За необхідності слід звернутися до оригінальної технічної документації від виробника процесору.

Добавлено примечание ([АЮ2]):

Шостий біт SF встановлюється в одиницю у разі переповнення або спустошення стека співпроцесора. Переповнення виникає у разі спроби запису у заповнений стек, а спустошення – у разі спроби читання з порожнього стека. Якщо прапорець SF встановлено, то значення прапорця C1=1 сигналізує про переповнення, а C1=0 – про спустошення стеку співпроцесора.

У молодших шести бітах SWR установленням відповідного біту в одиничне значення фіксується факт виникнення різних виключних ситуацій унаслідок виконання команд:

IE – неприпустима операція;

DE – денормалізований операнд;

ZE – помилка ділення на нуль;

OE – переповнення (результат дуже великий);

UE – антипереповнення (результат дуже малий);

PE – помилка точності (результат не може бути поданий точно).

Із виключеннями математичного співпроцесора також пов'язаний біт ES (біт 7) SWR і певна частина бітів іншого регістру – CWR (Control Word Register).

Молодші шість бітів CWR використовуються для маскування виключень. Установлення в одиницю будь-яких бітів PM, UM, OM, ZM, DM, IM маскує відповідні виключні ситуації PE, UE, OE, ZE, DE, IE. У разі виникнення виключної ситуації вона завжди фіксується встановленням в одиничне значення одного із шести молодших бітів SWR, однак під час цього також перевіряється відповідний біт маски в регістрі CWR. Якщо виключення незамасковане, то в регістрі SWR встановлюється в одиничне значення біт ES. За встановленого біта ES генерується виключення помилки арифметики з плаваючою комою (#MF) з номером 10h і виконується передавання керування обробнику виключення.

Додаткова інформація, яка може знадобитися обробнику виключення зберігається в регістрах IPR, DPR і OR. У регістрі OR зберігається одинадцять молодших біт (старші п'ять завжди 11011b) машинного коду команди, яка виконувалася на момент виникнення виключення, у регістрі IPR – її адреса, у регістрі DPR – адреса її операнда, якщо він розташований у пам'яті.

Питання апаратної реалізації системи переривань процесору розглядатимуться у межах навчального курсу «Архітектура комп'ютерів», а питання програмної обробки виключень – у межах курсу «Системне програмне забезпечення». Варто зазначити, що команда FINIT, із якої зазвичай починається робота з математичним співпроцесором, виконує ініціалізацію початкового стану регістру CWR так, що усі виключення виявляються замаскованими. Замасковані виключення обробляються самим математичним співпроцесором.

Обробка полягає у тому, що співпроцесор залежно від ситуації, розміщує в регістрах стека одне зі спеціальних числових значень.

Інші біти CWR використовуються для встановлення різних режимів обробки чисел з плаваючою комою. Біт IF залишився для забезпечення сумісності з математичним співпроцесором i80287, а двобітові поля RC та PC керують округленням і визначають точність обчислень.

Якщо число R *не може бути подане точно* в форматі, що визначається типом операнда для його розміщення, то визначаються найближчі до нього числа A і B, які відповідають вимогам формату і такі що A<R<B.

Наприклад, під мантису числа в короткому 32-бітному форматі виділяється 23 біти. Якщо R = 1.0001 0000 1000 0011 1001 011**1** E 101, то

A = 1.0001 0000 1000 0011 1001 011 E 101

B = 1.0001 0000 1000 0011 1001 100 E 101

Правила округлення числа визначає бітове поле RC:

00 – до найближчого з чисел A і B;

01 – до меншого, тобто R=A;

10 – до більшого, тобто R=B;

11 – округлення відкиданням дробової частини числа.

Значення RC=00 використовується за замовчуванням (установлюється командою FINIT).

Поле керування точністю дозволяє визначити довжину мантиси числа:

00 – довжина мантиси 24 біти;

10 – довжина мантиси 53 біти;

11 – довжина мантиси 64 біти.

Значення PC=11 відповідає «рідному» розширеному формату даних співпроцесора і використовується за замовчуванням (установлюється командою FINIT). Інші значення були введені для забезпечення сумісності зі стандартом IEEE 754 та специфікаціями деяких мов програмування.

7.1.3 Система команд співпроцесора

Вивчаючи систему команд математичного співпроцесора, їх зручно розподілити на кілька груп (Рисунок 7.3) і розглядати окремо за групами.

Рисунок 7.3 – Групи команд математичного співпроцесора

Кожна наступна група команд, зображених на рисунку зліва направо, розширюватиме можливості щодо програмування математичного співпроцесора.

7.1.3.1 Команди керування

Команди керування контролюють стан і визначають режими виконання команд математичним співпроцесором завдяки маніпуляціям з регістрами його програмної моделі. Зокрема ці команди дозволяють отримати доступ до регістрів CWR та SWR, які безпосередньо є програмно не доступними.

У групі команд керування існують парні команди, мнемонічне позначення яких відрізняється однією літерою. Одна з команд (F-команда) починається із звичного для усіх команд співпроцесора префікса «F», а інша (FN-команда) – із префікса «FN». Різниця цих команд полягає у тому, що перед виконанням F-команди перевіряється наявність будь-якого немаскованого виключення, а перед виконанням FN-команди – ні.

Команда F[N]INIT виконує ініціалізацію регістрів математичного співпроцесора значеннями (Таблиця 7.1), які відповідають їх початковому стану за замовчуванням.

Команда залишає незмінним вміст регістрів стека співпроцесора, але помічає їх як порожні (TAG=0FFFFh).

Таблиця 7.1

Регістр	CWR	SWR	TAG	IPR	OP	DPR
Значення	37Fh	0	0FFFFh	0	0	0

Команди F[N]STSW <приймач> вивантажують (копіюють) значення регістру SWR у <приймач>. Операнд <приймач> може бути словом у пам'яті або регістром AX.

$$\text{F[N]STSW <приймач>} = \begin{cases} \text{FSTSW m16} \\ \text{FNSTSW m16} \\ \text{FSTSW AX} \\ \text{FNSTSW AX} \end{cases}$$

Команди, які використовують як операнд регістр AX зручно використовувати для контролю стану обчислювального процесу і зміни порядку його розвитку за допомогою команд умовного передавання керування.

У системі команд математичного співпроцесора не передбачено ніяких спеціальних команд умовного передавання керування і для організації розгалужень використовуються команди основного процесора. Під час цього виникають певні незручності, бо виконання команд математичного співпроцесора впливає на біти його власного регістру прапорців SWR, а команди умовного передавання керування основного процесора реагують на стан бітів RFLAGS.

Після виконання команд FSTSW/FNSTSW AX стан будь-якого біта регістру прапорців SWR може бути визначений перевіркою значення відповідного йому біту в регістрі AX за допомогою команд основного процесора, і умовне передавання керування можна реалізувати за такими схемами:

FSTSW/FNSTSW AX	FSTSW/FNSTSW AX
TEST AX, маска	SAHF
Jcc <операнд>	Jcc <операнд>

Використана в другій схемі команда SAHF завантажує значення регістру AH, тобто значення восьми старших бітів SWR, у вісім молодших бітів регістру прапорців процесора RFLAGS і внаслідок цього значення біту C0 опиняється у CF, C2 у PF, C3 у ZF (команда SAHF обмежено підтримується в 64-бітному режимі).

Команда FSTCW/FNSTCW <приймач> завантажує значення CWR у слово, розміщене в пам'яті за адресою <приймач>. Команда FLDCW <джерело> завантажує в CWR значення слова, розміщеного в пам'яті за адресою <джерело>.

Команди FCLEX/FNCLEX встановлюють в нульове значення біти B, SF, ES, PE, UE, OE, ZE, DE в SWR.

Команди FSTENV/FNSTENV <операнд> зберігають у буфер пам'яті розміром не менше ніж 28 байтів за адресою, що визначається <операнд> вміст регістрів CWR, SWR, TAG, IPR, OP, DPR, а команда FLDENV <операнд> виконує дії, зворотні FSTENV/FNSTENV.

Команди FSAVE/FNSAVE <операнд> зберігають у буфер пам'яті розміром не менше ніж 108 байтів за адресою, що визначається <операнд> вміст регістрів CWR, SWR, TAG, IPR, OP, DPR, R0-R7 і після цього виконують реініціалізацію математичного співпроцесора, аналогічно тому, як це робить команда FINIT. Поновити попередньо збережений командами FSAVE/FNSAVE стан математичного співпроцесора можна командою FRSTOR <операнд>.

Команди FINCSTP/FDECSTP виконують інкремент/декремент поля TOP у SWR. Операції виконуються за правилами кільцевого буфера (Рисунок 7.2) і не впливають на вміст регістрів R0-R7 і TR. Наслідком команд є циклічне зрушення структури стека на одну позицію.

Команда FFREE ST(i) установлює в 11 (порожній) значення поля TR, відповідне регістру даних співпроцесора, зв'язаному з ST(i) не змінюючи вміст самого регістру і поля TOP у SWR.

7.1.3.2 Команди пересилання і обміну

Група команд пересилання і обміну надзвичайно важлива, оскільки робота з математичним співпроцесором без більшої частини команд цієї групи взагалі неможлива. Ці команди дозволяють завантажувати операнди з пам'яті для подальшої їх обробки з використанням команд математичного співпроцесора, зберігати результати обчислень в пам'яті та переміщувати значення поміж елементами стека співпроцесора. Команди пересилання і обміну, зображені на наступному рисунку, поділяються на чотири підгрупи.

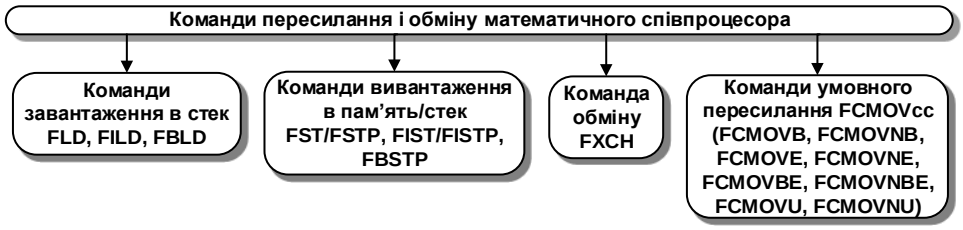


Рисунок 7.4 – Групи команд пересилання і обміну математичного співпроцесора

Команди $F\begin{bmatrix} I \\ B \end{bmatrix}$LD <джерело> завантажують (копіюють) у верхівку стека ST(0) значення, що визначається операндом <джерело>.

$$\begin{cases} \text{FLD джерело} \\ \text{FILD джерело} \\ \text{FBLD джерело} \end{cases} = \begin{cases} \text{FLD mem32fp/mem64fp/mem80fp/ST(i)} \\ \text{FILD mem16int/mem32int/mem64int} \\ \text{FBLD mem80bcd} \end{cases}$$

Команди $F\begin{bmatrix} I \\ B \end{bmatrix}$ST[P] <приймач> вивантажують значення з верхівки стека ST(0) у місце розташування, що визначається операндом <приймач>. Команди з постсуфіксом «P» після цього ще виштовхують значення з верхівки стека ST(0), водночас звільняючи стек.

$$\begin{cases} \text{FST приймач} \\ \text{FIST приймач} \\ \text{FSTP приймач} \\ \text{FISTP приймач} \\ \text{FBSTP приймач} \end{cases} = \begin{cases} \text{FST mem32fp/mem64fp/ST(i)} \\ \text{FIST mem16int/mem32int} \\ \text{FSTP mem32fp/mem64fp/mem80fp/ST(i)} \\ \text{FISTP mem16int/mem32int/mem64int} \\ \text{FBSTP mem80bcd} \end{cases}$$

Команди у формі FLD ST(i), FST ST(i) та FSTP ST(i) дозволяють виконувати пересилання даних поміж верхівкою та іншими елементами стека співпроцесора.

Команда FXCH [ST(i)] виконує обмін значення на верхівці стека співпроцесора з елементом стека ST(i) або ST(1), якщо операнд не зазначений. Комбінації цієї команди з різними операндами та/або з командами FLD/FST[P] ST(i) дозволяють маніпулювати стеком співпроцесора. Наприклад, обмін значень ST(1) і ST(2) може бути виконаний такою послідовністю команд:

```
fxch st(1)
fxch st(2)
fxch st(1)
```

Команди FCMOVcc ST(0),ST(i) завантажують (копіюють) у верхівку стека ST(0) значення елемента стека ST(i) унаслідок виконання умов, що визначаються значенням окремих бітів або комбінацій бітів регістру RFLAGS і позначаються мнемокодом:

$$\text{cc} = \begin{cases} \text{B} \\ \text{E} \\ \text{BE} \\ \text{U} \\ \text{NB} \\ \text{NE} \\ \text{NBE} \\ \text{NU} \end{cases} \text{ якщо } \begin{cases} \text{CF=1 (Below)} \\ \text{ZF=1 (Equal)} \\ \text{CF=1 або ZF=1 (Below or Equal)} \\ \text{PF=1 (Unordered)} \\ \text{CF=0 (Not Below)} \\ \text{ZF=0 (Not Equal)} \\ \text{CF=0 та ZF=0 (Not Below or Equal)} \\ \text{PF=0 (Not Unordered)} \end{cases}$$

Зазвичай ці команди використовуються відразу після команд порівняння, однак за допомогою пари команд FSTSW/FNSTSW AX та SAHF/TEST AX їх

можна використовувати і відразу після будь яких команд, виконання яких впливає на значення восьми старших бітів SWR, зокрема на прапорці C3, C2, C1, C0.

7.1.3.3 Базові арифметичні команди

До базових арифметичних команд, окрім команд для безпосереднього виконання чотирьох арифметичних операцій, належать також деякі додаткові команди і тому цю групу поділяють на п’ять підгруп (Рисунок 7.5).

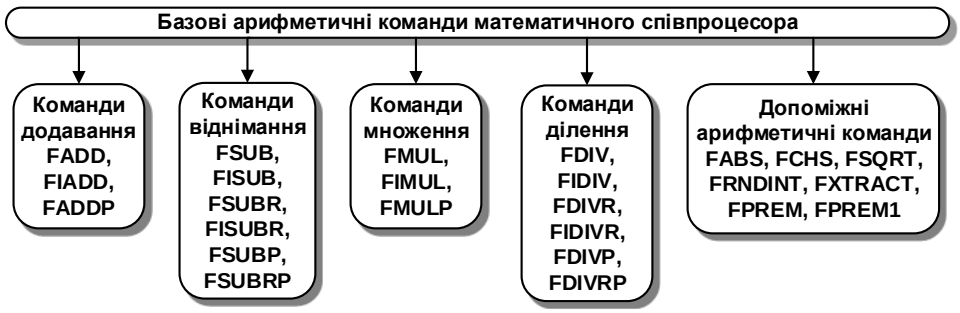


Рисунок 7.5 – Базові арифметичні команди FPU

Усі бінарні команди для виконання чотирьох арифметичних операцій подані командами у трьох форматах – без операндів, з одним і з двома операндами.

Команди додавання з одним операндом:

$$\text{F[I]ADD<операнд>}=\begin{cases}\text{FADD операнд}\\ \text{FIADD операнд}\end{cases}=\begin{cases}\text{FADD mem32fp/mem64fp}\\ \text{FIADD mem16int/mem32int}\end{cases}$$

додають значення <операнд> до значення на верхівці стека ST(0) і зберігають результат у ST(0).

Команди додавання з двома операндами

$$\text{FADD[P] [<операнд1>,<операнд2>]}=\begin{cases}\text{FADD ST(0),ST(i)}\\ \text{FADD ST(i),ST(0)}\\ \text{FADDP ST(i),ST(0)}\end{cases}$$

додають значення <операнд2> до значення <операнд1> і зберігають результат в <операнд1>. Команди з постсуфіксом «P» після цього ще виштовхують значення з верхівки стека ST(0), звільняючи стек. Команда без операндів еквівалентна команді FADDP ST(1),ST(0).

Команди віднімання з одним операндом

$$\text{F[I]SUB[R]<операнд>}=\begin{cases}\text{FSUB операнд}\\ \text{FISUB операнд}\\ \text{FSUBR операнд}\\ \text{FISUBR операнд}\end{cases}=\begin{cases}\text{FSUB mem32fp/mem64fp}\\ \text{FISUB mem16int/mem32int}\\ \text{FSUBR mem32fp/mem64fp}\\ \text{FISUBR mem16int/mem32int}\end{cases}$$

без постсуфікса «R» віднімають значення <операнд> від значення на верхівці стека ST(0) і зберігають результат у ST(0). Постсуфікс «R» змінює порядок операндів, тобто команди віднімають значення на верхівці стека ST(0) від значення <операнд> і зберігають результат у ST(0).

Команди віднімання з двома операндами

$$\text{FSUB[R][P] [<операнд1>,<операнд2>]}=\begin{cases}\text{FSUB ST(0),ST(i)}\\ \text{FSUB ST(i),ST(0)}\\ \text{FSUBP ST(i),ST(0)}\\ \text{FSUBR ST(0),ST(i)}\\ \text{FSUBR ST(i),ST(0)}\\ \text{FSUBRP ST(i),ST(0)}\end{cases}$$

без постсуфікса «R» віднімають значення <операнд2> від значення <операнд1> і зберігають результат в <операнд1>. Команди з постсуфіксом «R» віднімають значення <операнд1> від значення <операнд2> і зберігають результат в <операнд1>. Команди з постсуфіксом «P» після цього ще виштовхують значення з верхівки стека ST(0), звільняючи стек. Команда без операндів FSUB еквівалентна команді FSUBP ST(1),ST(0), а команда без операндів FSUBR еквівалентна команді FSUBRP ST(1),ST(0).

Команди множення з одним операндом

$$F[I]MUL<операнд>=\begin{cases}FMUL\ операнд\\FIMUL\ операнд\end{cases}=\begin{cases}FMUL\ mem32fp/mem64fp\\FIMUL\ mem16int/mem32int\end{cases}$$

множать значення <операнд> на значення на верхівці стека ST(0) і зберігають результат у ST(0).

Команди множення з двома операндами

$$FMUL[P]\ [<операнд1>,<операнд2>]=\begin{cases}FMUL\ \ ST(0),ST(i)\\FMUL\ \ ST(i),ST(0)\\FMULP\ \ ST(i),ST(0)\end{cases}$$

множать значення <операнд2> на значення <операнд1> і зберігають результат в <операнд1>. Команди з постсуфіксом «P» після цього ще виштовхують значення з верхівки стека ST(0), звільняючи стек. Команда без операндів еквівалентна команді FMULP ST(1),ST(0).

Команди ділення з одним операндом

$$F[I]DIV[R]<операнд>=\begin{cases}FDIV\ операнд\\FIDIV\ операнд\\FDIVR\ операнд\\FIDIVR\ операнд\end{cases}=\begin{cases}FDIV\ mem32fp/mem64fp\\FIDIV\ mem16int/mem32int\\FDIVR\ mem32fp/mem64fp\\FIDIVR\ mem16int/mem32int\end{cases}$$

без постсуфікса «R» ділять значення на верхівці стека ST(0) на значення <операнд> і зберігають результат у ST(0). Постсуфікс «R» змінює порядок операндів, тобто команди ділять значення <операнд> на значення на верхівці стека ST(0) і зберігають результат у ST(0).

Команди ділення з двома операндами

$$FDIV[R][P]\ [<операнд1>,<операнд2>]=\begin{cases}FDIV\ ST(0),ST(i)\\FDIV\ ST(i),ST(0)\\FDIVP\ ST(i),ST(0)\\FDIVR\ ST(0),ST(i)\\FDIVR\ ST(i),ST(0)\\FDIVRP\ ST(i),ST(0)\end{cases}$$

без постсуфікса «R» ділять значення <операнд1> на значення <операнд2> і зберігають результат в <операнд1>.

Команди з постсуфіксом «R» ділять значення <операнд2> на значення <операнд1> і зберігають результат в <операнд1>.

Команди з постсуфіксом «P» після цього ще виштовхують значення з верхівки стека ST(0), звільняючи стек. Команда без операндів FDIV еквівалентна команді FDIVP ST(1),ST(0), а команда без операндів FDIVR еквівалентна команді FDIVRP ST(1),ST(0).

Серед допоміжних арифметичних команд варто назвати тільки чотири – FABS, FCHS, FSQRT і FRNDINT. Призначення інших допоміжних арифметичних команд поки що може виявитися незрозумілим, і тому будуть розглянуті в контексті інших задач, у яких вони виявляються корисними.

Команда FABS замінює число на верхівці стека ST(0) його абсолютним значенням (модулем). Команда FCHS змінює знак числа на верхівці стека ST(0) на протилежний. Команда FSQRT обчислює квадратний корінь числа на верхівці стека ST(0) і розміщує результат у ST(0). Команда FRNDINT округлює значення числа в ST(0) до цілого числа відповідно до режиму округлення (заданим бітами RC CWR).

Команд пересилання і обміну та базових арифметичних команд уже достатньо для реалізації деяких чисельних алгоритмів, що і демонструють наведені далі приклади. Приклади обмежуються використанням тільки тих функцій, обчислення яких можна виконати на підставі основних арифметичних команд. Це обмеження ніяк не впливає на реалізацію алгоритму, і в подальшому приклади можуть бути легко пристосовані для розв’язування інших задач, заміною коду для обчислення значення функції з використанням трансцендентних функцій.

Справа після символу коментаря ";" наводиться значення регістрів стека арифметичного співпроцесора після виконання команд починаючи з логічної верхівки стека ST(0).

Приклад 7.1. Чисельне інтегрування $\int_0^{10} x^2 dx = \frac{x^3}{3}\Big|_0^{10} = 333.3(3)$ методом прямокутників $\int_a^b f(x)dx = \frac{b-a}{n}\sum_{i=0}^{n-1} f\left(a + \frac{b-a}{n}\cdot i\right)$.

```
include \masm64\include64\masm64rt.inc
.data
integral    REAL8 0.0
a           REAL8 0.0
b           REAL8 10.0
n           DWORD 100000
.code
main        proc
        call AllocConsole
        finit
        fld b               ; b
        fsub a              ; b-a
        fidiv n             ; h=(b-a)/n
        fld a               ; a, h
        fld st(0)           ; a, a, h
        call func           ; s=f(a), a, h
        fxch                ; a, s, h
        mov ebx,n
        dec ebx
@@:     fadd st(0),st(2)    ; x=a+h, s, h
        fld st(0)           ; x, x, s, h
        call func           ; f(x), x, s, h
        faddp st(2),st(0)   ; x, s=s+f(a+h), h
        dec ebx
```

```
        or ebx,ebx
        jnz @B
        fxch st(2)          ; h, s=s+f(a+h), x
        fmul                ; I=s*h, x
        fstp integral       ; x
        fstp st(0)          ;
        rcall vc_printf,cfm$("%s%f"),cfm$("Integral = "),integral
        waitkey cfm$("\nPress \lEnter\r ...")
        ret
main endp
func proc
; Вхід – число в st(0), ; Вихід – квадрат числа в st(0)
        fld st(0)
        fmul
        ret
func endp
end
```

Приклад 7.2. Чисельне інтегрування $\int_0^{10} 3x^2 dx = 3 \cdot \frac{x^3}{3}\Big|_0^{10} = x^3\Big|_0^{10} = 1000$

методом Гауса $\int_a^b f(x)dx = \left[x = \frac{a+b}{2} + \frac{b-a}{2} t \right] = \int_{-1}^{1} f(t)dt = \frac{b-a}{2} \sum_{i=1}^{n} A_i f(t_i) + \Delta$,

де $\Delta = \frac{(b-a)^{2n+1}(n!)^4 M_{2n}}{\left[(2n)!\right]^3 (2n+1)}$, M_{2n} – максимальне значення 2n-ї похідної на [a,b].

A_i, t_i для n=8 наведені в таблиці.

i	t_i	A_i	i	t_i	A_i
1	-0.96028996	0.10122854	5	+0.18343464	0.36268378
2	-0.79666648	0.22238104	6	+0.52553142	0.31370664

3	-0.52553142	0.31370664	7	+0.79666648	0.22238104
4	-0.18343464	0.36268378	8	+0.96028996	0.10122854

```
; Файл IntegralGauss.inc
include \masm64\include64\masm64rt.inc
.const
t       REAL8 -0.96028996, -0.79666648, -0.52553142, -0.18343464
        REAL8  0.18343464, 0.52553142, 0.79666648, 0.96028986
A       REAL8 0.10122854, 0.22238104, 0.31370664, 0.36268378
        REAL8 0.36268378, 0.31370664, 0.22238104, 0.10122854
.data
integral        REAL8 0.0
a               REAL8 0.0
b               REAL8 10.0
; Файл IntegralGauss.asm
include IntegralGauss.inc
.code
main   proc
        call AllocConsole
        mov ecx,8
        xor rdi,rdi
        finit
        fldz                ; s=0
        fld1                ; 1, s
        fld1                ; 1, 1, s
        fadd                ; 2, s
        fld b               ; b, 2, s
        fadd a              ; b+a, 2, s
        fdiv st(0),st(1)    ; (b+a)/2, 2, s
```

```
        fld b               ; b, (b+a)/2, 2, s
        fsub a              ; b-a , (b+a)/2, 2, s
        fdiv st(0),st(2)    ; (b-a)/2 , (b+a)/2, 2, s
@@:     fld st(0)           ; (b-a)/2, (b-a)/2 , (b+a)/2, 2, s
        fld t[rdi]          ; t, (b-a)/2 , (b-a)/2 ,(b+a)/2, 2, s
        fmul                ; t*(b-a)/2 , (b-a)/2 ,(b+a)/2, 2, s
        fadd st(0),st(2)    ; x=t*(b-a)/2 + (b+a)/2, (b-a)/2 ,(b+a)/2, 2, s
        call func           ; f(x), (b-a)/2 ,(b+a)/2, 2, s
        fmul A[rdi]         ; A*f(x), (b-a)/2 ,(b+a)/2, 2, s
        faddp st(4),st(0)   ; (b-a)/2 ,(b+a)/2, 2, s=s+A*f(x)
        add rdi,8
        loop @B
        fmul st(0),st(3)    ; s*(b-a)/2, (b+a)/2, 2, s
        fstp integral       ; (b+a)/2, 2, s
        rcall vc_printf,cfm$("%s%f"),cfm$("Integral = "),integral
        waitkey cfm$("\nPress \lEnter\r ...")
        ret
main   endp
func proc ; Вхід – число в st(0), ; Вихід – 3*квадрат числа в st(0)
        fld st(0)
        fmul
        fld1
        fld1
        fld1
        fadd
        fadd
        fmul
        ret
func endp
```

end

Приклад 7.3. Розв'язок задачі Коші $\frac{dy}{dx} = 2(x^2 + y), y(0) = 1$ методом Рунге – Кутта 4-го порядку. Значення невідомої функції y в точці x_{n+1} відносно значення в попередній точці x_n для задачі Коші $y' = f(x, y), y(x_0) = y_0$ обчислюється методом Рунге–Кутта 4-го порядку за формулою

$y_{n+1} = y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$, $x_{n+1} = x_n + h$, де

$k_1 = f(x_n, y_n),\ k_2 = f(x_n + \frac{h}{2}, y_n + \frac{h}{2}k_1),$

$k_3 = f(x_n + \frac{h}{2}, y_n + \frac{h}{2}k_2),\ k_4 = f(x_n + h, y_n + hk_3)$

```
; Файл RungeKutt4.inc
include \masm64\include64\masm64rt.inc
k=20
.data
xn                  dq 0.0
yn                  dq 1.0
xk                  dq 1.0
n                   dd k
.data?
h                   dq ?
h2                  dq ?
h6                  dq ?
rk1                 dq ?
rk2                 dq ?
rk3                 dq ?
rk4                 dq ?
x                   dq k dup(?)
y                   dq k dup(?)
```

```
; Файл RungeKutt4.asm
include RungeKutt4.inc
.code
main   proc
LOCAL Buffer[4096]:BYTE
        call AllocConsole
        mov ebx,n
        xor rdi,rdi
        finit
        fld yn              ; yn
        fld xn              ; xn, yn
        fld xk              ; xk, xn, yn
        fsub st(0),st(1)    ; xk-xn, xn, yn
        fild n              ; n, xn-xk, xn, yn
        fdivp st(1),st(0)   ; h=(xn-xk)/n, xn, yn
        fld st(0)           ; h, h, xn, yn
        fld st(0)           ; h, h, h, xn, yn
        fld1                ; 1, h, h, h, xn, yn
        fld1                ; 1, 1, h, h, h, xn, yn
        fadd                ; 2, h, h, h, xn, yn
        fld st(0)           ; 2, 2, h, h, h, xn, yn
        fadd st(1),st(0)    ; 2, 4, h, h, h,  xn, yn
        fadd st(1),st(0)    ; 2, 6, h, h, h, xn, yn
        fdivp st(2),st(0)   ; 6, h/2, h, h, xn, yn
        fdivp st(2),st(0)   ; h/2, h/6, h, xn, yn
        fstp h2             ; h/6, h, xn, yn
        fstp h6             ; h, xn, yn
        fstp h              ; xn, yn
@@:  fld st(0)              ; xn, xn, yn
```

```
fxch st(2)          ; yn, xn, xn
fld st(0)           ; yn, yn, xn, xn
fxch st(3)          ; xn, yn, xn, yn
call func           ; rk1=f(xn,yn), xn, yn
fst rk1
fld h2              ; h/2, rk1, xn, yn
fmul st(1),st(0)    ; h/2, rk1*h/2, xn, yn
fadd st(0),st(2)    ; xn+h/2, rk1*h/2, xn, yn
fxch                ; rk1*h/2, xn+h/2, xn, yn
fadd st(0),st(3)    ; yn+rk1*h/2, xn+h/2, xn, yn
fxch                ; xn+h/2, yn+rk1*h/2, xn, yn
call func           ; rk2=f(xn+h/2, yn+rk1*h/2), xn, yn
fst rk2             ; rk2, xn, yn
fld h2              ; h/2, rk2, xn, yn
fmul st(1),st(0)    ; h/2, rk2*h/2, xn, yn
fadd st(0),st(2)    ; xn+h/2, rk2*h/2, xn, yn
fxch                ; rk2*h/2, xn+h/2, xn, yn
fadd st(0),st(3)    ; yn+rk2*h/2, xn+h/2, xn, yn
fxch                ; xn+h/2, yn+rk2*h/2, xn, yn
call func           ; rk3=f(xn+h/2, yn+rk2*h/2), xn, yn
fst rk3             ; rk3, xn, yn
fld h               ; h, rk3, xn, yn
fmul st(1),st(0)    ; h, h*rk3, xn, yn
fadd st(0),st(2)    ; xn+h, h*rk3, xn, yn
fxch                ; h*rk3, xn+h, xn, yn
fadd st(0),st(3)    ; yn+h*rk3, xn+h, xn, yn
fxch                ; xn+h, yn+h*rk3, xn, yn
call func           ; rk4=f(xn+h, yn+h*rk3), xn, yn
fld rk3             ; rk3, rk4, xn, yn
```

```
        fld rk2             ; rk2, rk3, rk4, xn, yn
        fld rk1             ; rk1, rk2, rk3, rk4, xn, yn
        faddp st(3),st(0)   ; rk2, rk3, rk4+rk1, xn, yn
        fld st(0)           ; rk2, rk2, rk3, rk4+rk1, xn, yn
        fadd                ; 2*rk2=rk2+rk2, rk3, rk4+rk1, xn, yn
        faddp st(2),st(0)   ; rk3, rk4+2*rk2+rk1, xn, yn
        fld st(0)           ; rk3, rk3, rk4+2*rk2+rk1, xn, yn
        fadd                ; 2*rk3=rk3+rk3, rk4+2*rk2+rk1, xn, yn
        fadd                ; rk4+2*rk3+2*rk2+rk1,xn, yn
        fmul h6             ; h*(rk4+2*rk3+2*rk2+rk1)/6,xn, yn
        faddp st(2),st(0)   ; xn, yi=yn+h*(rk4+2*rk3+2*rk2+rk1)/6
        fadd h              ; xi=xn+h, yi=yn+h*(rk4+2*rk3+2*rk2+rk1)/6
        fst x[rdi*8]
        fxch
        fst y[rdi*8]
        fxch
        inc rdi
        dec ebx
        or ebx,ebx
        jnz @B
        mov ebx,n
        xor rdi,rdi
        lea rsi,Buffer
@@:     mov dword ptr [rsi],"(f=y"
        lea rsi,[rsi+4]
        mov r12,x[rdi*8]
        rcall fptoa,r12,rsi
        rcall lstrlen,rsi
        mov word ptr [rsi+rax],"=)"
```

```
        lea rsi,[rsi+rax+2]
        mov r12,y[rdi*8]
        rcall fptoa,r12,rsi
        rcall lstrlen,rsi
        mov dword ptr [rsi+rax],0A0Dh
        lea rsi,Buffer
        invoke StdOut,rsi
        inc rdi
        dec ebx
        or ebx,ebx
        jnz @B
        waitkey cfm$("\nPress \lEnter\r ...")
        ret
main   endp
func proc     ; dy/dx=f(x,y)=2*(x*x+y)
; Вхід  : x в st(0), y в st(1)       ; x, y
; Вихід : f(x,y) в st(0)
                fld st(0)             ; x, x, y
                fmul                  ; x*x, y
                fadd                  ; x*x+y
                fld st(0)             ; x*x+y, x*x+y
                fadd                  ; 2*(x*x+y)
                ret
func   endp
end
```

7.1.3.4 Команди обчислення трансцендентних функцій

Система команд математичного співпроцесора має усього дев'ять команд (Рисунок 7.6) для обчислення трансцендентних функцій, які можна поділити на три підгрупи.

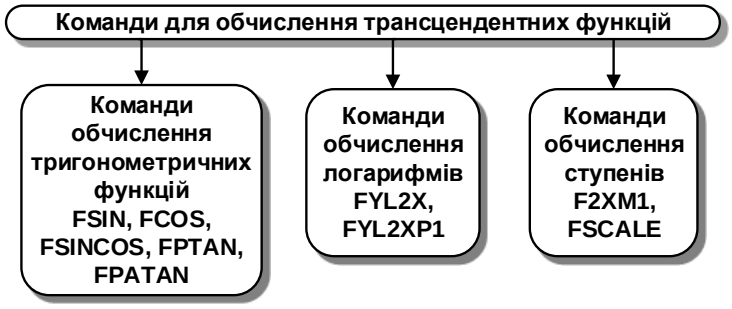


Рисунок 7.6 – Команди обчислення трансцендентних функцій FPU

Команди FSIN, FCOS, FSINCOS, FPTAN використовують як операнд значення в ST(0). Операнд вважається заданим у радіанах і повинен бути в діапазоні від -2^{63} до 2^{63}. Якщо значення операнда виходить за цей діапазон, команди встановлюють в 1 прапор C2 в SWR, залишаючи незмінним стан стека і значення в ST(0).

Команди FSIN та FCOS замінюють значення операнда в ST(0) значенням, відповідно, синуса та косинуса операнда.

Команда FSINFCOS обчислює значення синуса та косинуса операнда в ST(0), замінює значення операнда в ST(0) значенням синуса операнда, а після цього заносить значення косинуса операнда на верхівку стека. Після виконання команди в ST(1) розміщується синус, а в ST(0) – косинус операнда.

Команда FPTAN обчислює значення тангенса операнда в ST(0), замінює значення операнда в ST(0) значенням тангенса операнда, а після цього заносить значення одиниці на верхівку стека. Після виконання команди в ST(0) розміщується 1.0, а в ST(1) – тангенс операнда. Одиниця в ST(0) розміщується

для того, щоб можна було обчислити котангенс як $\mathrm{ctg(x)}=\frac{1}{\mathrm{tg(x)}}$ за допомогою команди FDIVR відразу за командою FPTAN.

Команда FPATAN обчислює значення арктангенса числа, отриманого внаслідок ділення значення в ST(1) на значення в ST(0), зберігає результат у ST(1) і виштовхує ST(0) зі стека. Результат завжди має такий самий знак, як і ST(1), і значення менше за абсолютною величиною, ніж число π.

Для обчислення арктангенса числа потрібно спочатку завантажити на стек співпроцесора число, потім одиницю, а потім скористатися командою FPATAN, тобто як $\mathrm{arctg}\left(\mathrm{x}\right)=\mathrm{arctg}\left(\frac{\mathrm{x}}{1}\right)$.

Дещо дивний підхід щодо операндів команди FPATAN має геометричну інтерпретацію: команда обчислює кут між віссю абсцис OX і лінією, проведеною з центру координат O(0,0) у точку з координатами M(X,Y)=M(ST(0),ST(1)).

За умови відсутності команд для обчислення інших зворотних тригонометричних функцій, обчислення командою FPATAN арктангенса від $\frac{\mathrm{ST(1)}}{\mathrm{ST(0)}}$ зрозуміло якщо згадати, що:

$$\arcsin(z)=arctg\left(\frac{z}{\sqrt{1-z^2}}\right)=\left[\begin{matrix}y=z\\x=\sqrt{1-z^2}\end{matrix}\right]=arctg\left(\frac{y}{x}\right)$$

$$\arccos(z)=arctg\left(\frac{\sqrt{1-z^2}}{z}\right)=\left[\begin{matrix}y=\sqrt{1-z^2}\\x=z\end{matrix}\right]=arctg\left(\frac{y}{x}\right)$$

$$\arccos(z)=2arctg\left(\sqrt{\frac{1-z}{1+z}}\right)=\left[\begin{matrix}y=\sqrt{1-z}\\x=\sqrt{1+z}\end{matrix}\right]=2arctg\left(\frac{y}{x}\right)$$

Команда FYL2X обчислює значення $\mathrm{ST(1)}\cdot\log_2\mathrm{ST(0)}$, зберігає результат у ST(1) і виштовхує ST(0) зі стека. *Операнд у ST(0) повинен бути більший, ніж нульове значення*.

Команда FYL2XP1 обчислює значення $ST(1)\cdot\log_2(ST(0)+1.0)$, зберігає результат у ST(1) і виштовхує ST(0) зі стека. Операнд у ST(0) повинен бути від $-\left(1-\frac{\sqrt{2}}{2}\right)$ до $\left(1-\frac{\sqrt{2}}{2}\right)$. Операнд у ST(1) може бути будь-яким числом.

Команда F2XM1 зводить 2 до ступеня ST(0) і від цього значення віднімає 1. Результат зберігається в ST(0): $ST(0) \leftarrow 2^{ST(0)}-1$. Значення ST(0) повинно лежати в межах від -1 до +1, інакше результат невизначений. За такого обмеження на значення ST(0), значення функції $f(x)=2^x$ наближені до одиниці. Вирахування одиниці закладено в алгоритм команди для підвищення точності, бо в такому випадку в мантисі результату *після нормалізації* зберігається більше значущих цифр.

Призначення останньої команди FSCALE з групи команд для обчислення трансцендентних функцій буде розглянуто у наступному розділі після вивчення команд завантаження констант.

7.1.3.5 Команди завантаження констант

Команди завантаження констант можна було б віднести до групи команд пересилання і обміну, однак призначення деяких з них було б абсолютно не очевидним без попереднього розгляду команд для обчислення трансцендентних функцій. Ця група налічує 7 команд, які завантажують на верхівку стека ST(0) значення числових констант (Таблиця 7.2), аналогічно команді FLD.

Таблиця 7.2

Команда	FLDZ	FLD1	FLDPI	FLDL2T	FLDL2E	FLDLG2	FLDLN2
Константа	+0.0	+1.0	π	$\log_2 10$	$\log_2 e$	$\log_{10} 2$	$\log_e 2$
Значення	0	1	$\approx 3{,}14$	$\approx 3{,}32$	$\approx 1{,}44$	$\approx 0{,}30$	$\approx 0{,}69$

Константи подано у внутрішньому форматі числа з плаваючою комою подвійної точності, що забезпечує 19 десяткових цифр. Третій рядок таблиці

наведено лише з метою приблизного визначення діапазону, у який потрапляють константи.

Команди завантаження констант, у поєднанні з командами для обчислення ступенів двійки F2XM1/FSCALE і командами для обчислення логарифмів FYL2X/FYL2XP1, розширюють можливості системи команд співпроцесора щодо обчислення інших трансцендентних функцій.

Обчислення функцій виду $f(x) = z^x$ виконується з використанням співвідношення $z^x = 2^{x\log_2 z}$ за такою схемою:

$$2^x = \left(2^x - 1\right) + 1 => F2XM1(x) + 1$$

$$e^x = \left(2^{x\log_2 e} - 1\right) + 1 => F2XM1(x \cdot \log_2 e) + 1 => F2XM1(x \cdot FLDL2E) + 1$$

$$10^x = \left(2^{x\log_2 10} - 1\right) + 1 => F2XM1(x \cdot \log_2 10) + 1 => F2XM1(x \cdot FLDL2T) + 1$$

$$z^x = \left(2^{x\log_2 z} - 1\right) + 1 => F2XM1(x \cdot \log_2 z) + 1 => F2XM1(FYL2X(x, z)) + 1.$$

Наведенні вище формули пояснюють призначення констант, що завантажуються командами FLDL2T і FLDL2E, однак щодо цього виникає одне питання.

Команда F2XM1 обчислює значення функції $f(x) = 2^x - 1$ тільки для $-1 < x < 1$, інакше результат невизначений. Чисельні значення констант $\log_2 10$ і $\log_2 e$ (Таблиця 7.2) уже перевищують значення одиниці, а наведені формули передбачають множення цих констант іще і на значення аргументу функції. Як у такому випадку обчислити, наприклад, значення $f(x) = e^x$ або власне саму $f(x) = 2^x$ для будь якого x>1?

Такі обмежені можливості команди F2XM1 можна обійти за допомогою команди FSCALE і співвідношення $2^{x+y} = 2^x \cdot 2^y$. Команда FSCALE *спочатку округлює до цілого числа* значення ST(1), а потім множить значення ST(0) на два в ступені ST(1) і записує результат у ST(0): $ST(0) \leftarrow ST(0) \cdot 2^{[ST(1)]}$.

Обчислення функцій виду $f(x)=2^x$ для $|x|>1$ виконується за такою схемою:

$2^x = 2^{[x]+(x-[x])} = 2^{x-[x]} \cdot 2^{[x]} = \left(\left(2^{x-[x]}-1\right)+1\right) \cdot 2^{[x]}$, де [ ] – взяття цілої частини

$=> FSCALE(F2XM1(x-FRNDINT(x))+1, x)$.

Приклад 7.4. Підпрограма обчислення $y=e^x$. Числа x та y зберігаються в пам'яті у довгому форматі числа з плаваючою комою і передаються за адресою. Підпрограма повністю зберігає контекст математичного співпроцесора і повертає в молодшому слові RAX вміст SWR на момент виконання останньої команди математичного співпроцесора *до відновлення* його контексту.

```
include \masm64\include64\masm64rt.inc
.data
x       QWORD    1.0
.data?
y       QWORD    ?
.code
exponenta           proc
LOCAL FpuContext:WOW64_FLOATING_SAVE_AREA
LOCAL CWR:WORD
        fsave FpuContext
        fld qword ptr [rcx]         ; x
        fldl2e                      ; log2 e , x
        fmul                        ; x · log2 e
        fld   st(0)                 ; x · log2 e , x · log2 e
        fstcw CWR
        mov ax,CWR
        mov cx,ax
        or ah,1100b                 ; RC=11
```

```
        mov CWR,ax
        fldcw CWR
        frndint                 ; [x·log2 e],  x·log2 e
        mov CWR,cx
        fldcw CWR
        fsubr  st(0),st(1)      ; x·log2 e−[x·log2 e],  x·log2 e
        f2xm1                   ; 2^(x·log2 e−[x·log2 e]) −1,  x·log2 e
        fld1                    ; 1, 2^(x·log2 e−[x·log2 e]) −1,  x·log2 e
        fadd                    ; 2^(x·log2 e−[x·log2 e]),  x·log2 e
        fscale                  ; 2^(x·log2 e−[x·log2 e]) · 2^[x·log2 e]
        fstsw ax                ; (R|E)AX для повернення з функції
        fstp  qword ptr [rdx]
        frstor FpuContext
        ret
exponenta endp
main        proc
        lea rcx,x
        lea rdx,y
        call exponenta
        add rsp,2*8
        rcall vc_printf,cfm$("%s%f"),cfm$("e^x = "),y
        waitkey cfm$("\nPress \lEnter\r ...")
        ret
main endp
end
```

Обчислення функцій виду $f(x)=\log_a b$ $(a>0, a\neq 1, b>0)$ виконується з використанням команди FYL2X, констант (команд) $\log_{10} 2$ (FLDLG2) і $\log_e 2$ (FLDLN2) та співвідношення $\log_a x = \log_a 2 \cdot \log_2 x$ за такою схемою:

$\log_2 x => FYL2X(1, x)$

$\lg x = \log_{10} x = \log_{10} 2 \cdot \log_2 x => FYL2X(\log_{10} 2, x) => FYL2X(FLDLG2, x)$

$\ln x = \log_e x = \log_e 2 \cdot \log_2 x => FYL2X(\log_e 2, x) => FYL2X(FLDLN2, x)$

$$\log_a b = \frac{\log_c b}{\log_c a} = \frac{\log_2 b}{\log_2 a} => \frac{FYL2X(1, b)}{FYL2X(1, a)}.$$

7.1.3.6 Команди порівняння і аналізу даних

Команди FCOM[P][P] [<операнд>] порівнюють значення ST(0) з <операнд> і встановлюють прапорці C0, C2, C3 в SWR відповідно до результату порівняння як зазначено в таблиці 7.3.

Таблиця 7.3

FCOM[P][P]	C3	C2	C0	FICOM[P]	C3	C2	C0
ST(0) > операнд	0	0	0	ST(0) > операнд	0	0	0
ST(0) < операнд	1	0	0	ST(0) < операнд	0	0	1
ST(0) = операнд	0	0	1	ST(0) = операнд	1	0	0
Не зрівнянні	1	1	1	Не зрівнянні	1	1	1

FCOM[P][P] [<операнд>]= { FCOM mem32fp/mem64fp; FCOM ST(i); FCOM; FCOMP mem32fp/mem64fp; FCOMP ST(i); FCOMP; FCOMPP

Команди з постсуфіксом «P» після цього ще виштовхують значення з верхівки стека ST(0), а команда з постсуфіксом «PP» двічі виштовхує значення

ST(0), тобто обидва значення в ST(0) та ST(1). Команди без операндів еквівалентні командам з операндами в ST(0) та ST(1).

Команди FICOM[P] <операнд>=$\begin{cases}\text{FICOM mem16int/mem32int}\\\text{FICOMP mem16int/mem32int}\end{cases}$ порівнюють значення ST(0) з *цілим числом* у пам'яті за адресою, що визначається <операнд> і встановлюють прапорці C0, C2, C3 в SWR відповідно до результату порівняння (Таблиця 7.3). Команди з постсуфіксом «P» після цього ще виштовхують значення із ST(0), звільняючи таким чином стек.

Подальший аналіз значень прапорців C0, C2, C3 в SWR після виконання команд порівняння FCOM[P][P]/FICOM[P] і реалізація умовного передавання керування виконується за допомогою команди FSTSW за схемою, наведеною в розділі 7.1.3.1.

Команди FCOMI[P] ST(0),ST(i)=$\begin{cases}\text{FCOMI ST(0),ST(i)}\\\text{FCOMIP ST(0),ST(i)}\end{cases}$ мають менші можливості щодо місця розташування і типів операндів, однак у багатьох випадках їх використання для реалізації розгалужень виявляється зручнішим, бо може виконуватися безпосередньо після виконання команд без використання команди FSTSW та інших.

Ці команди порівнюють значення ST(0) з ST(i) і встановлюють прапорці ZF, PF, CF в RFLAGS відповідно до результату порівняння (Таблиця 7.4). Команда FCOMIP після цього ще виштовхує значення із ST(0), звільняючи стек.

Таблиця 7.4

FCOMI[P]	ZF	PF	CF	FTST	C3	C2	C0
ST(0) > ST(i)	0	0	0	ST(0) > 0.0	0	0	0
ST(0) < ST(i)	0	0	1	ST(0) < 0.0	0	0	1
ST(0) = ST(i)	1	0	0	ST(0) = 0.0	1	0	0
Незрівнянні	1	1	1	Незрівнянні	1	1	1

Команда FTST порівнює значення ST(0) з нулем і встановлює прапорці C0, C2, C3 в SWR відповідно до результату порівняння (Таблиця 7.4). під час порівняння знак нуля ігнорується як $-0.0 \leftarrow +0.0$.

Використання команд порівняння і реалізацію розгалужень демонструє приклад.

Приклад 7.5. Визначення кореня рівняння $\cos x - \sqrt{x} = 0$ на сегменті $\left[0, \frac{\pi}{2}\right]$ методом дихотомії (поділу навпіл інтервалу невизначеності).

$$f(a) = \cos 0 - \sqrt{0} = 1 - 0 > 0, \quad f(b) = \cos\frac{\pi}{2} - \sqrt{\frac{\pi}{2}} = 0 - \sqrt{\frac{\pi}{2}} < 0$$

```
include \masm64\include64\masm64rt.inc
.data
eps             REAL8 1.0E-15
a               REAL8 0.0
.data?
b               REAL8 ?
x               REAL8 ?
fx              REAL8 ?
.code
main proc
        finit
        fld eps         ; eps
        fld1            ; 1, eps
        fld1            ; 1, 1, eps
        fadd            ; 2,eps
        fld a           ; a,2,eps
        fld st(1)       ; 2,a,2,eps
        fldpi           ; pi,2,a,2,eps
        fdivr           ; b=pi/2,a,2,eps
```

```
next:                   ; b,a,2,eps
        fld st(0)       ; b,b,a,2,eps
        fsub st(0),st(2)  ; b-a,b,a,2,eps
        fabs            ;|b-a|,b,a,2,eps
        fcomip st(0),st(4)  ; b,a,2,eps
        jc OK           ; st(0)<st(i) => ZF=PF=0, CF=1
        fld st(0)       ; b,b,a,2,eps
        fadd st(0),st(2)  ; b+a,b,a,2,eps
        fdiv st(0),st(3)  ; c=(b+a)/2,b,a,2,eps
        fld st(0)       ; c,c,b,a,2,eps
        call Function   ; f(c),c,b,a,2,eps
        ftst            ; f(c),c,b,a,2,eps
        fstsw ax        ; B C3 TOP C2 C1 C0
        test ax,4500h   ; 0   1  000   1  0    1   00000000b
        fstp st(0)      ; c,b,a,2,eps
        jz @F           ; C3=C2=C0=0 => st(0)>0
        fstp st(2)      ; b,a=c,2,eps
        jmp next        ; b,a,2,eps
@@:     fstp st(1)      ; b=c,a,2,eps
        jmp next        ; b,a,2,eps
OK:     fadd            ; b+a,2,eps
        fdivr           ; x=(b+a)/2,eps
        fld st(0)       ; x,x,eps
        call Function   ; f(x),x,eps
        fstp fx         ; x,eps
        fstp x          ; eps
        fstp st(0)
        rcall vc_printf,cfm$("%s%f"),cfm$("x="),x
        rcall vc_printf,cfm$("%s%f"),cfm$("\nf\ax\b="),fx
```

```
        waitkey cfm$("\nPress \lEnter\r ...")
        ret
main endp
Function      proc   ; f(x)= cos(x)-sqrt(x)
; Вхід  : x в st(0), вихід : f(x) в st(0)
              fld st(0)
              fsqrt
              fxch
              fcos
              fsub
              ret
Function      endp
End
```

8 ПРОГРАМУВАННЯ РОЗШИРЕННЯ MMX

Розширення MMX мікропроцесорів Intel64/AMD64 являє собою програмно-апаратне доповнення архітектури мікропроцесора, що надає їй нові властивості, які забезпечують більш високу продуктивність мультимедійних додатків у системах обробки і передавання даних. Уперше це розширення з'явилося 1997 року в мікропроцесорах Intel© Pentium MMX™, а надалі стало стандартом для усіх наступних моделей мікропроцесорів та набуло подальший розвиток.

8.1 Програмна модель розширення ММХ

Програмною моделлю розширення ММХ традиційно вважають його організацію щодо можливостей програмування.

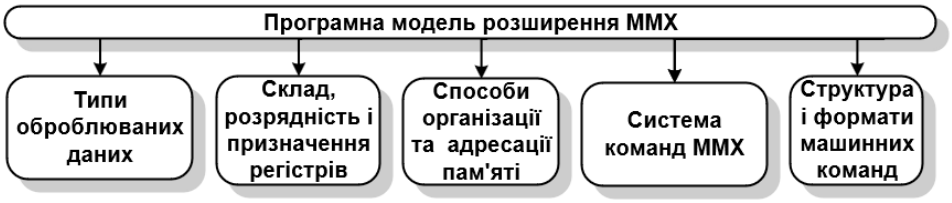


Рисунок 8.1 відображає складові програмної моделі розширення ММХ.

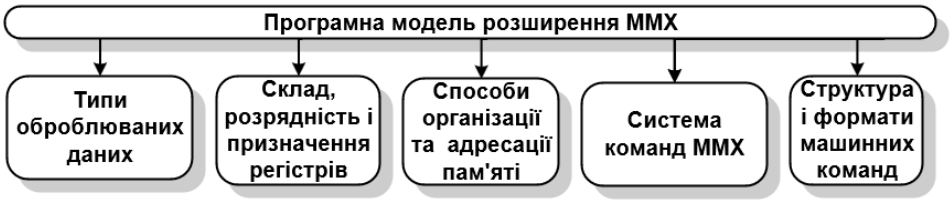


Рисунок 8.1 – Програмна модель розширення ММХ

Ураховуючи, що ММХ є розширенням, тобто складовою частиною основного процесора, у контексті його програмної моделі не розглядаються режими роботи, які воно «успадковує» від основного процесора і тому вони не зображені на рисунку.

Оскільки розширення ММХ є складовою частиною основного процесора, способи організації та адресації пам’яті для них не розрізняються. Вони подані на рисунку тому, що в системі команд ММХ є команди, які передбачають використання операндів у пам’яті і для них використовуються відомі з попередніх розділів способи адресації.

8.2 Типи оброблюваних даних

На *апаратному рівні* ММХ-команди підтримують єдиний тип даних – 64-розрядні зчетверені слова, однак *логічна інтерпретація* цього апаратно

підтримуваного типу в залежності від команди може бути різна. Рисунок 8.2 демонструє усі типи даних розширення MMX.

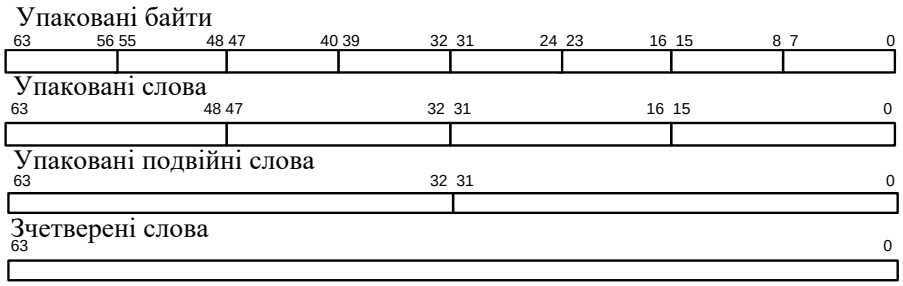


Рисунок 8.2 – Типи даних розширення MMX

Характерною ознакою MMX-розширення є *векторність*, тобто 64-розрядний пакет бітів інтерпретується як набір декількох компонентів (координат вектора), які можуть оброблятися MMX-командами *паралельно і не залежно* одна від одної. Такі векторні типи даних називають пакованими, а компоненти векторів у MMX-розширенні інтерпретуються тільки як *знакові або беззнакові цілі числа*.

8.3 Склад, розрядність і призначення регістрів

Як апаратний ресурс для обробки 64-розрядних операндів у розширенні MMX використовуються 64 молодші біти 80-розрядних регістрів стека математичного співпроцесора R0-R7 (

Рисунок 8.3).

Команди розширення MMX використовують регістри R0-R7 не як апаратний стек, а як звичайну регістрову пам’ять з довільним доступом, звертаючись до регістрів за іменами MM0-MM7. Жоден з регістрів MMX не має певного спеціалізованого призначення і всі вони використовуються як апаратний ресурс для обробки даних.

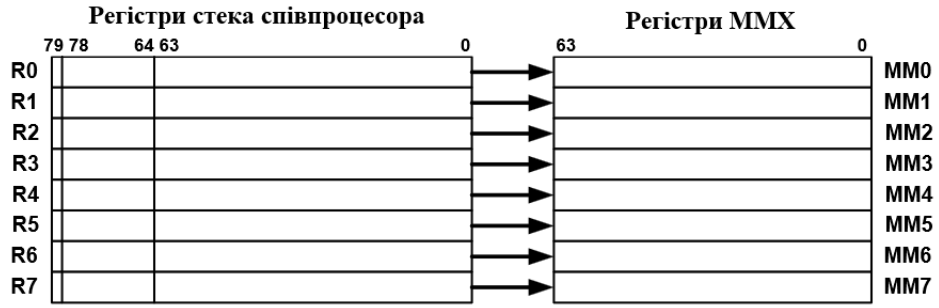


Рисунок 8.3 – Схема використання регістрів стека розширенням MMX

Оскільки математичний співпроцесор і розширення MMX використовують спільні апаратні ресурси, використання в коді *поєднання* команд FPU і команд MMX майже неможливе, але вони можуть використовуватися *по черзі в окремих блоках* зі збереженням, за необхідності, результатів роботи окремих блоків у пам’яті. Блок MMX-команд повинен завершуватися командою без операндів EMMS.

8.4 Система команд MMX

Для вивчення системи команд MMX ми традиційно розподілимо їх на декілька груп (Рисунок 8.4), і розглядатимемо окремо за групами. Кожна наступна група команд, зображених на рисунку зліва направо, розширюватиме можливості програмування.

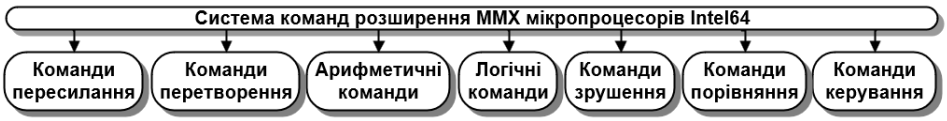


Рисунок 8.4 – Система команд ММХ

Перш ніж безпосередньо розглядати систему команд розширення ММХ, слід зробити деякі зауваження щодо їх загальних особливостей. Рисунок 8.5 демонструє принцип векторної обробки, характерної для більшості ММХ-команд.

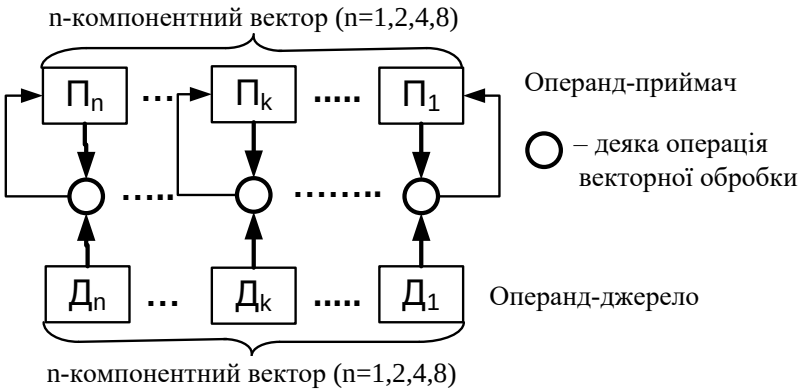


Рисунок 8.5 – Схема векторної обробки даних

Команда обробки виконує попарно визначену операцію «○» паралельно і незалежно з відповідними елементами вектор-операндів і розміщує результат у відповідних елементах одного з операндів: $П_i \leftarrow П_i \circ Д_i$ (i=1..n).

Особливістю арифметичних команд і деяких команд перетворення є те, що вони виконують обробку або за правилами циклічної арифметики (wraparound arithmetic), або за правилами арифметики з насиченням (saturation arithmetic).

За правилами циклічної арифметики (арифметики з циклічним перенесенням), якщо довжина розрядної сітки, потрібної для розміщення результату операції, перевищує довжину розрядної сітки вихідних операндів, то біти результату, що виходять за розрядну сітку, відкидаються.

За правилами арифметики з насиченням, якщо результат операції перевищує максимально/мінімально можливе значення для подання вихідних

операндів, то як результат операції приймається це максимальне/мінімальне значення. Для цього слід пам'ятати про те, що максимально і мінімально можливе значення подання *для цілих зі знаком і без знаку відрізняються*.

8.4.1 Команди пересилання

Команди пересилання дозволяють завантажувати операнди з пам'яті для подальшої їх обробки з використанням MMX-команд, зберігати результати обробки в пам'яті, переміщувати дані поміж MMX-регістрами та поміж MMX-регістрами і регістрами загального призначення:

$$\text{MOV}\begin{pmatrix}\text{D}\\ \text{Q}\end{pmatrix}\ \text{<приймач>,<джерело>}=\begin{cases}\text{MOVD mm,r32/m32}\\ \text{MOVD r32/m32,mm}\\ \text{MOVQ mm,r64/m64}\\ \text{MOVQ r64/m64,mm}\end{cases}$$

Команда MOVD працює тільки з 32-ма молодшими розрядами MMX-регістра і якщо MMX-регістр є приймачем, то його старші 32 біта заповнюються нулями.

8.4.2 Команди перетворення

В групі команд перетворення типів можна виокремити дві підгрупи – команди упаковування і команди розпаковування. Наявність команд цієї групи пов'язана з особливостями виконання арифметичних MMX-команд.

До команд упаковування належать PACKSSWB, PACKUSWB і PACKSSDW. Рисунок 8.6 демонструє загальний алгоритм виконання команд PACKSSWB/PACKUSWB з узагальненим позначенням

$$\text{PACK}\begin{pmatrix}\text{SS}\\ \text{US}\end{pmatrix}\text{WB <Приймач>,<Джерело>}=\begin{cases}\text{PACKSSWB mm,mm}\\ \text{PACKSSWB mm,m64}\\ \text{PACKUSWB mm,mm}\\ \text{PACKUSWB mm,m64}\end{cases}.$$

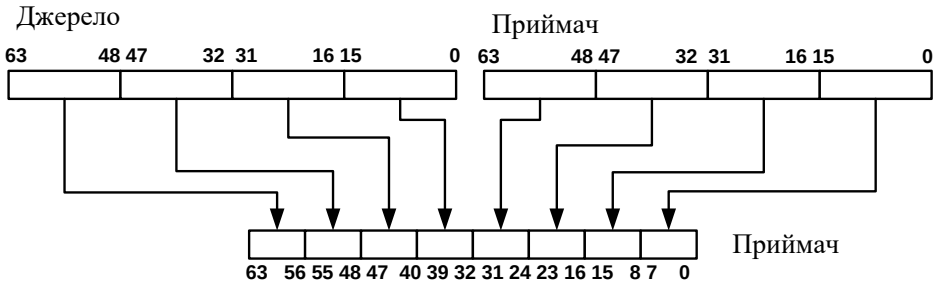


Рисунок 8.6 – Загальний алгоритм виконання команд PACKSSWB/PACKUSWB

Команди PACKSSWB і PACKUSWB відрізняються лише тим, як вони інтерпретують компоненти векторів операндів у вихідному стані. Команда PACKSSWB виконує пакування зі знаковим насиченням, а команда PACKUSWB – із беззнаковим.

Команда *PACKSSDW mm,mm / m*64 виконує пакування зі знаковим насиченням за алгоритмом, аналогічним командам PACKSSWB і PACKUSWB

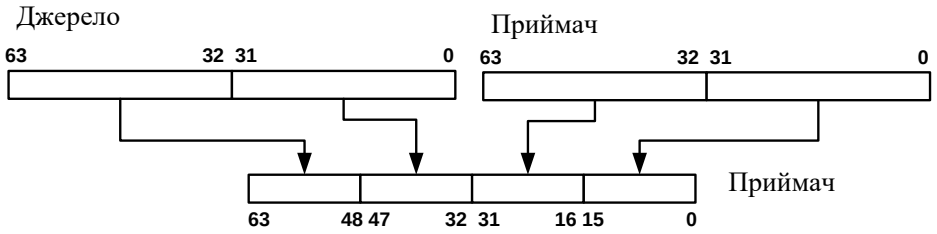


Рисунок 8.7 – Схема алгоритму виконання команди PACKSSDW

Узагальнене позначення команд розпаковування має такий вигляд:

$$\text{PUNPCK}\begin{pmatrix}\text{H}\\\text{L}\end{pmatrix}\begin{pmatrix}\text{BW}\\\text{WD}\\\text{DQ}\end{pmatrix}\ \text{<Приймач>,<Джерело>}=\begin{cases}\text{PUNPCKHBW mm,mm/m64}\\\text{PUNPCKHWD mm,mm/m64}\\\text{PUNPCKHDQ mm,mm/m64}\\\text{PUNPCKLBW mm,mm/m64}\\\text{PUNPCKLWD mm,mm/m64}\\\text{PUNPCKLDQ mm,mm/m64}\end{cases}$$

Командам цієї групи не притаманні особливості ні циклічної, ні арифметики з насиченням і вони просто виконують переміщення компонентів векторів операндів. Рисунок 8.8 демонструє алгоритм виконання команди PUNPCKHBW, а Рисунок 8.9 – команди PUNPCKLBW.

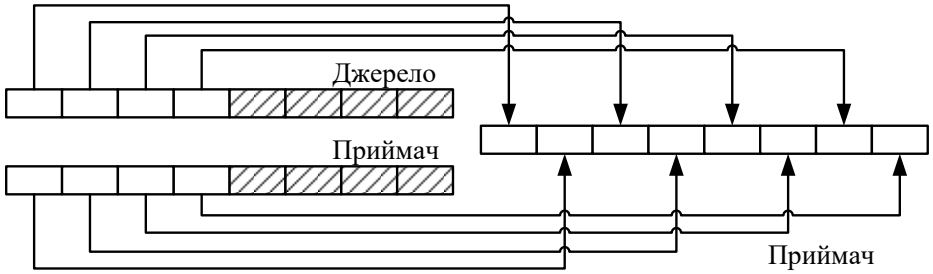


Рисунок 8.8 – Алгоритм виконання команди PUNPCKHBW

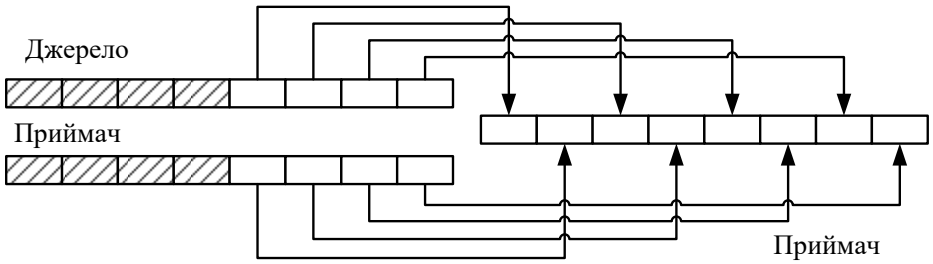


Рисунок 8.9 – Алгоритм виконання команди PUNPCKLBW

Алгоритми інших команд розпаковування аналогічні вищенаведеним. Вони збирають подвійні слова зі слів (WD) і зчетверене слово з подвійних слів

(DQ), вибираючи потрібні компоненти зі старшої (H) або молодшої (L) половини операндів.

8.4.3 Арифметичні команди

Команди з узагальненим позначенням

$$P\begin{pmatrix}\text{ADD}\\ \text{SUB}\end{pmatrix}\begin{pmatrix}\text{B}\\ \text{W}\\ \text{D}\end{pmatrix}\ \text{<приймач>,<джерело>}$$

виконують операції додавання та віднімання з відповідними елементами приймача і джерела *за правилами циклічної арифметики* та розміщують результат у відповідних елементах приймача. Символи «B», «W» і «D» у мнемонічному позначенні команди указують на тип компонентів – байт, слово і подвійне слово, відповідно. Приймачем може бути тільки MMX-регістр, а джерело може розміщуватися як у MMX-регістрі, так і в пам'яті.

Команди з узагальненим позначенням

$$P\begin{pmatrix}\text{ADD}\\ \text{SUB}\end{pmatrix}\begin{pmatrix}\text{S}\\ \text{US}\end{pmatrix}\begin{pmatrix}\text{B}\\ \text{W}\end{pmatrix}\ \text{<приймач>,<джерело>}$$

виконують операції додавання та віднімання з відповідними елементами приймача і джерела за правилами арифметики зі знаковим (S) та беззнаковим (US) насиченням і розміщують результат у відповідних елементах приймача. Символи «B» і «W» у мнемонічному позначенні команди указубть на тип компонентів – байт і слово, відповідно. Приймачем може бути тільки MMX-регістр, а джерело може розміщуватися як у MMX-регістрі, так і в пам'яті.

Команди з узагальненим позначенням

$$\text{PMUL}\begin{pmatrix}\text{H}\\ \text{L}\end{pmatrix}\text{W<приймач>,<джерело>}$$

виконують попарне *знакове* множення *слів* приймача та джерела і розрізняються лише тим, як зберігають результат. Команда із символом «L» у мнемонічному позначенні зберігає у відповідних словах приймача 16 молодших, а команда із

символом «H» – 16 старших розрядів результату. Приймачем може бути тільки MMX-регістр, а джерело може розміщуватися як у MMX-регістрі, так і в пам'яті.

Команда PMADDWD mm,mm/m64 спочатку попарно множить слова операндів, потім попарно додає отримані внаслідок множення суміжні подвійні слова і розміщує результат на місце лівого операнду.

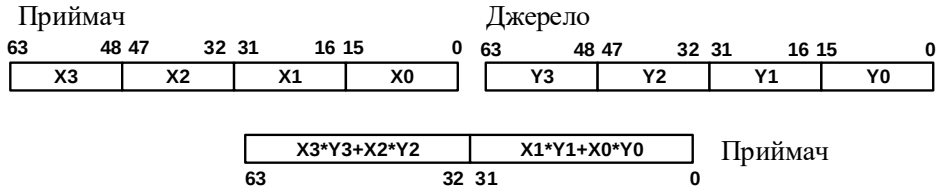


Рисунок 8.10 – Схема розміщення результатів команди PMADDWD

Команд пересилання, перетворення типів та арифметичних команд вже виявляється достатньо для реалізації деяких алгоритмів, що і демонструє наступний приклад.

Приклад 8.1. Консольний додаток contrast.exe для лінійного контрастування зображення 24-бітового bmp-файлу. Як параметр командного рядка приймає ім'я вхідного bmp-файлу і сторює вихідний файл output.bmp.

За лінійного контрастування використовується лінійне поелементне перетворення виду y=ax+b, параметри a і b якого визначаються бажаними значеннями мінімальної y_{min} і максимальної y_{max} вихідної яскравості.

Очевидно, що за лінійного перетворення мінімальному x_{min} і максимальному x_{max} значенню вхідного сигналу відповідатиме мінімальне y_{min} і максимальне y_{max} значення вихідного.

Унаслідок зміни y_{min} і y_{max} бажаними фіксованими значеннями мінімальної та максимальної вихідної яскравості і за умови, що x_{min} і x_{max} – це фіксовані характеристики вхідного сигналу, отримуємо систему двох лінійних рівнянь з двома невідомими a і b:

$$\begin{cases} y_{min} = a \cdot x_{min} + b \\ y_{max} = a \cdot x_{max} + b \end{cases}$$

Розв'язавши відносно a і b систему і підставивши в y=ax+b отримаємо формулу лінійного контрастування $y = \frac{y_{max} - y_{min}}{x_{max} - x_{min}} \cdot (x - x_{min}) + y_{min}$.

Файли бітових образів починаються зі структури

```
BITMAPFILEHEADER STRUCT
    bfType          WORD ?     ; сигнатура «BM» (42h 4Dh)
    bfSize          DWORD ?    ; розмір файлу
    bfReserved1     WORD ?     ; не використовується
    bfReserved2     WORD ?     ; не використовується
    bfOffBits       DWORD ?    ; зсув даних бітового образу від заголовка
BITMAPFILEHEADER ENDS
```

Безпосередньо за нею розташовується структура BITMAPINFO, що містить усю інформацію про бітовий образ:

```
BITMAPINFO STRUCT
  bmiHeader     BITMAPINFOHEADER <?>;
  bmiColors     RGBQUAD 1 dup (<>);
BITMAPINFO ENDS
```

Структура BITMAPINFO ділиться на дві частини: масив структур RGBQUAD, що містить дані бітового образу і структуру BITMAPINFOHEADER, що описує характеристики бітового образу.

```
RGBQUAD STRUCT
    rgbBlue         BYTE ? ; червона складова
    rgbGreen        BYTE ? ; зелена складова
    rgbRed          BYTE ? ; синя складова
    rgbReserved     BYTE ? ; альфа-канал
RGBQUAD ENDS
```

Загалом структура BITMAPINFOHEADER залежить від версії, однак перші її елементи фіксовані:

```
BITMAPINFOHEADER STRUCT
    biSize          DWORD ?  ; розмір цієї структури
    biWidth         LONG ?   ; ширина
    biHeight        LONG ?   ; і висота рисунку у пікселях
    biPlanes        WORD ?
    biBitCount      WORD ?   ; кількість бітів на піксель
    biCompression   DWORD ?
    biSizeImage     DWORD ?
    biXPelsPerMeter LONG ?
    biYPelsPerMeter LONG ?
    biClrUsed       DWORD ?
    biClrImportant  DWORD ?
BITMAPINFOHEADER ENDS
```

Для зручності роботи з файлами в коді використовується наданий операційною системою механізм файлів, відображених у пам'ять. Як тільки файл відображено в пам'ять, до нього можна звертатися так, начебто він цілком до неї завантажений. Отже, робота з файлом, відображеним у пам'ять, виконується як з масивом байтів без виконання операцій вводу/виводу. Для файлу, відображеного в пам'ять, операційна система забезпечує тотожність вмісту дискового файлу й області пам'яті, на яку він відображений. Тобто, коли відображення знімається, вміст відображення фіксується у файлі.

```
; Файл contrast.asm
include \masm64\include64\masm64rt.inc
include contrast.inc      ; на жаль у заголовних файлах MASM64 SDK відсутній
; опис деяких структур. У файлі contrast.inc визначені вищенаведені структури
; BITMAPFILEHEADER, BITMAPINFO, RGBQUAD, BITMAPINFOHEADER
; У розширенні MMX немає команди ділення.
```

```
; "MMX-ділення" реалізуємо власним макровизначенням pdivuslb
; на базі основної системи команд
pdivuslb      macro RegMMX1,RegMMX2,RegMMX3
        push rax      ; зберігаємо
        push rbx      ; усі регістри
        push rcx      ; які використовуються
        push rdx      ; у макровизначенні
        movd eax,RegMMX1 ; перекладаємо ммх-регістри
        movd ebx,RegMMX2 ; у регістри загального призначення
; EAX буде надавати байти для ділення а EBX дільники
        mov rcx,4    ; будуть побайтно ділитися 4 байти
@@:  push rax      ; RAX і RBX будуть зіпсовані внаслідок першого ділення
        push rbx      ; зберігаємо для подальших ітерацій циклу
        and eax,0FFh        ;виділити слово для діленням
        and ebx,0FFh        ;виділити молодший байт
        div bl              ; ділення AX/BL
        shrd edx,eax,8      ; зберегти результат ділення AL в EDX
        pop rbx             ; поновити попередні
        pop rax             ; значення регістрів
        shr eax,8           ; підготовути в AL і BL
        shr ebx,8           ; наступні байти для ділення
        loop @B
        movd RegMMX3,edx ; зберігаємо результат у ммх-регістрі
        pop rdx       ; поновлюємо
        pop rcx       ; усі
        pop rbx       ; регістри
        pop rax
endm
.const
```

```
Ymin            DWORD 0
Ymax            DWORD 00FFFFFFh
.data?
argv            LPCWSTR ?; потрібні для отримання параметра командого
argc            DWORD ?  ; рядка за допомогою CommandLineToArgvW
hFile1          HANDLE ? ; хендл вхідного файла
hFile2          HANDLE ? ; хендл вихідного файла
1fileSizeLow    DWORD ?  ; розмір вхідного файла
hMemFile1       HANDLE ? ; хендл відображення вхідного файла
hMemFile2       HANDLE ? ; хендл відображення вихідного файла
hMem1           HANDLE ? ; адреса відображення вхідного файла
hMem2           HANDLE ? ; адреса відображення вихідного файла
.code
main  proc
        call AllocConsole
        call GetCommandLineW
        lea rbx,argc
        mov argv, rvcall(CommandLineToArgvW, rax, rbx);
        .if argv==0
                errorui()
                ret
        .endif
        mov rbx,argv
        mov rbx,[rbx+8]    ; в rbx адреса рядка з іменем вхідного файла
        invoke CreateFileW, rbx, GENERIC_READ, 0, 0,\
                                        OPEN_EXISTING, 0, 0
        mov hFile1,rax
        rcall LocalFree, argv
        .if hFile1 == INVALID_HANDLE_VALUE
```

```
        errorui()
        ret
.endif
; bfSize типу DWORD в BITMAPFILEHEADER
rcall GetFileSize, hFile1, NULL
.if fileSizeLow == -1 ; INVALID_FILE_SIZE
        errorui()
        jmp err1
.endif
mov fileSizeLow, eax
invoke CreateFileMapping, hFile1, NULL, \
                PAGE_READONLY, NULL, eax, NULL
.if rax == NULL
        errorui()
        jmp err1
.endif
mov hMemFile1, rax
invoke MapViewOfFile, rax, FILE_MAP_READ, NULL, NULL, NULL
.if rax == NULL
        errorui()
        jmp err2
.endif
mov hMem1,rax
mov rsi,rax  ; В rsi адреса початку даних файла-джерела
mov rbx,rax
lea rbx,[rbx+sizeof BITMAPFILEHEADER]
cmp word ptr (BITMAPINFO PTR [rbx]).bmiHeader.biBitCount, 24
je @F
;Цей формат BMP-файлу не підтримується
```

```
        rcall MessageBox,0,"Unsupported file format",0, MB_ICONERROR
        jmp err6
@@:  invoke CreateFile, chr$("output.bmp"),\
                    GENERIC_READ or GENERIC_WRITE, NULL, NULL,\
                    CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL
        .if rax == INVALID_HANDLE_VALUE
                errorui()
                jmp err3
        .endif
        mov hFile2, rax
        invoke CreateFileMapping, rax, NULL,\
                    PAGE_READWRITE, NULL, fileSizeLow, NULL
        .if rax == NULL
                errorui()
                jmp err4
        .endif
        mov hMemFile2, rax
        invoke MapViewOfFile, rax, FILE_MAP_WRITE, \
                                NULL, NULL, fileSizeLow
        .if rax == NULL
                errorui()
                jmp err5
        .endif
        mov hMem2,rax
        mov rdi,rax  ; В RDI адреса початку даних файла-приймача
; В RSI адреса початку даних файла-джерела (див. код вище)
; копіюємо заголовок у новий файл – він буде такий самий
        mov ecx,sizeof BITMAPFILEHEADER
        add ecx,(BITMAPINFO PTR [rbx]).bmiHeader.biSize
```

```
        rep movsb
        mov r12, rsi        ; потім знадобиться
;визначимо кількість пікселів для обробки як Width x Height
        mov eax,(BITMAPINFO PTR [rbx]).bmiHeader.biWidth
        mul (BITMAPINFO PTR [rbx]).bmiHeader.biHeight
        mov ecx,eax; bfSize в BITMAPFILEHEADER типу DWORD
        mov r13d, ecx       ; потім знадобиться
        dec ecx             ; на один піксель менше для пропуску
                            ; початкового значення min і max
; R1 G1 B1 R2 G2 B2 R3 G3 B3 . . .
        movd mm0,dword ptr [rsi]      ; початкове
        movq mm1,mm0                  ; значення min і max
@@:  add rsi,3                        ; беремо RGB
        movd mm2,dword ptr [rsi]      ; наступного пікселя
        pminub      mm0,mm2           ; і шукаємо
        pmaxub      mm1,mm2           ; min і max
        loop @B
        psubusb     mm1,mm0  ; Xmax - Xmin в mm1, Xmin в mm0
                             ; Ymin=0
        movd mm2,Ymax        ; Ymax - Ymin в mm2
; Вираз (Ymax - Ymin)/(Xmax - Xmin) складається з
; фіксованих значень і обчислюється тільки один раз
        pdivuslb mm2,mm1,mm3       ;mm3 = (Ymax - Ymin)/(Xmax - Xmin)
        mov ecx, r13d              ; кількість пікселів для обробки
        mov rsi, r12               ; в RSI адреса початку даних,
        ; а RDI залишилось від rep movsb, Xmin в mm0
        psubd   mm2,mm2      ; mm2=0 для розпакування
        punpcklbw mm3,mm2    ; розпакували для pmullw
@@:  movd   mm4,dword ptr [rsi]
```

```
        psubusb mm4,mm0        ; mm4=(X - Xmin)
        punpcklbw mm4,mm2    ; розпакували для pmullw
        pmullw        mm4,mm3  ; mm4=(X - Xmin)*(Ymax - Ymin)/(Xmax - Xmin)
        packuswb     mm4,mm2  ; запакували MM4 для пересилання
        movd  dword ptr [rdi],mm4
        add rsi,3      ; перерахунок RSI  та RDI
        add rdi,3      ; для зміщення на наступний RGB
        loop @B
err6:   rcall UnmapViewOfFile, hMem2
err5    :rcall CloseHandle, hMemFile2
err4    :rcall CloseHandle, hFile2
err3:   rcall UnmapViewOfFile, hMem1
err2:   rcall CloseHandle, hMemFile1
err1:   rcall CloseHandle, hFile1
        ret
main endp
end
```

Результати роботи програми цього прикладу демонструють такі рисунки.

Вхідне зображення

Вихідне зображення

Вхідне зображення

Вихідне зображення

8.4.4 Логічні команди

Характерною ознакою логічних команд є те, що їх можна вважати векторними, однак по суті їх виконання зводиться до побітових логічних операцій з усіма бітами операндів, так само, як і для логічних команд загального призначення:

PAND mm,mm/m64 – побітова кон'юнкція;

PANDN mm,mm/m64 – заперечення побітової кон'юнкції;

POR mm,mm/m64 – побітова диз'юнкція;

PXOR mm,mm/m64 – побітова сума за модулем два.

Серед логічних MMX-команд нема інверсії, однак її можна виконати скориставшись тим, що $\overline{x \cdot x} = \bar{x}$, тобто звести до виконання команди PANDN.

8.4.5 Команди зрушення

Серед команд зрушення виділяють команди логічних і арифметичних зрушень. Команди з узагальненим позначенням

$$PS\begin{pmatrix} L \\ R \end{pmatrix} L\begin{pmatrix} W \\ D \\ Q \end{pmatrix}$$ <приймач>,<джерело>

виконують *логічне* зрушення вправо або вліво (L|R) *усіх* слів|подвійних слів|зчетверених слів (W|D|Q) приймача на *однакову* кількість бітів, що визначається значенням джерела. Розряди, що звільняються унаслідок логічного зрушення заповнюються нулями.

Приймач може бути тільки MMX-регістром, а джерело – MMX-регістром або 64-розрядним операндом у пам'яті. Для усіх команд, окрім PSLLW, джерело також може задаватися безпосереднім 8-бітним значенням, яке визначає *однакову* кількість бітів для зрушення *усіх* елементів приймача.

Команди з узагальненим позначенням $\mathrm{PSRA}\begin{pmatrix}\mathrm{W}\\ \mathrm{D}\end{pmatrix}$ виконують *арифметичне* зрушення вправо *усіх* слів|подвійних слів (W|D) приймача на *однакову* кількість бітів, що визначається значенням джерела. Розряди що звільняються унаслідок логічного зрушення заповнюються значенням старшого біту відповідного елемента.

Приймач може бути тільки MMX-регістром, а джерело – MMX-регістром, 64-розрядним операндом у пам'яті або задаватися безпосереднім 8-бітним значенням.

8.4.6 Команди порівняння

Команди з узагальненим позначенням

$$\mathrm{PCMP}\begin{pmatrix}\mathrm{EQ}\\ \mathrm{GT}\end{pmatrix}\begin{pmatrix}\mathrm{B}\\ \mathrm{W}\\ \mathrm{D}\end{pmatrix}$$ <приймач>,<джерело>

виконують попарне порівняння відповідних байтів|слів|подвійних слів (B|W|D) приймача і джерела. Якщо відповідний елемент приймача і джерела однакові (EQ) або відповідний елемент приймача більший, ніж елемент джерела (GT), то елемент приймача заповнюється одиницями, інакше нулями.

9 КОНТРОЛЬНІ ПИТАННЯ

1. Текстові макро.

2. Макровизначення.

3. Вбудовані макрофункції @SizeStr, @InStr, @CatStr, @SubStr.

4. Рядкові оператори SIZESTR, INSTR, CATSTR, SUBSTR.

5. Макроблоки повторення FOR, FORC, REPEAT, WHILE.

6. Оператори співвідношення EQ, NE, LT, LE, GT, GE.

7. Логічні оператори NOT, AND, OR, XOR.

8. Директиви умовної трансляції IF/IFE, IFDEF/IFDNDEF, IFB/IFNB, IFIDN/IFDIF, IFIDNI/IFDIFI.

9. Директиви умовної генерації помилок .ERRE, .ERRNZ, .ERRDEF .ERRNDEF, .ERRB, .ERRNB, .ERRDIF, .ERRDIFI, .ERRIDNI.

10. Програмна модель математичного співпроцесора.

11. Типи даних математичного співпроцесора.

12. Склад і призначення регістрів математичного співпроцесора.

13. Система команд математичного співпроцесора.

13.1. Команди керування F[N]INIT, F[N]STSW, FSTCW/FNSTCW, FCLEX/FNCLEX, FSTENV/FNSTENV, FSAVE/FNSAVE, FRSTOR, FINCSTP/FDECSTP, FFREE.

13.2. Команди пересилання і обміну.

13.2.1. Команди завантаження FLD, FILD, FBLD.

13.2.2. Команди вивантаження FST/FSTP, FIST/FISTP, FBSTP.

13.2.3. Команда обміну FXCH.

13.2.4. Команди умовного пересилання FCMOVB, FCMOVNB, FCMOVE, FCMOVNE, FCMOVBE, FCMOVNBE, FCMOVU, FCMOVNU.

13.3. Базові арифметичні команди.

13.3.1. Додавання FADD, FIADD, FADDP.

13.3.2. Віднімання FSUB, FISUB, FSUBR, FISUBR, FSUBP, FSUBPR.

13.3.3. Множення FMUL, FIMUL, FMULP.

13.3.4. Ділення FDIV, FIDIV, FDIVR, FIDIVR, FDIVP, FDIVRP.

13.3.5. Допоміжні FABS, FCHS, FSQRT, FRNDINT, FXTRACT, FPREM, FPREM1.

13.4. Команди обчислення трансцендентних функцій.

13.4.1. Тригонометричних функцій FSIN, FCOS, FSINCOS, FPTAN.

13.4.2. Логарифмів FYL2X, FYL2XP1.

13.4.3. Ступенів F2XM, FSCALE.

13.5. Команди завантаження констант FLDZ, FLD1, FLDPI, FLDL2T, FLDL2E, FLDLG2, FLDLN2.

13.6. Команди порівняння і аналізу даних FCOM, FCOMP, FCOMPP, FICOM, FICOMP, FCOMI, FCOMIP, FTST.

14. Програмна модель розширення MMX.

15. Типи оброблюваних даних.

16. Склад, розрядність і призначення регістрів.

17. Система команд MMX:

17.1. Команди пересилання.

17.2. Команди перетворення.

17.3. Арифметичні команди.

17.4. Логічні команди.

17.5. Команди зрушення.

17.6. Команди порівняння.

10 ТЕСТОВІ ЗАВДАННЯ

1. Якого режиму роботи процесора не існує:

а) Режим реальних адрес; б) Захищений режим;

в) Графічний режим; г) Режим системного керування?

2. У якому режимі починає працювати процесор після вмикання живлення:

а) режим реальних адрес; б) захищений режим

в) режим системного управління; г) режим віртуального 8086?

3. Із якого режиму процесор не може безпосередньо перейти в режим віртуального 8086:

а) режим системного управління; б) режим реальних адрес;

в) захищений режим; г) режим сумісності?

4. Кількість бітів у байті дорівнює…

а) 2; б) 4; в) 8; г) 10.

5. Кількість бітів у слові дорівнює…

а) 4; б) 8; в) 10; г) 16.

6. Кількість бітів у подвійному слові дорівнює…

а) 10; б) 20; в) 32; г) 48.

7. Який діапазон відповідає 8-розрядному беззнаковому цілому:

а) від -128 до +217; б) від -128 до +217;

в) від 0 до 255; г) від -126 до +218?

8. Яка з даних команд копіює слово з регістру в комірку пам’яті:

а) stosw; б) lodsd; в) scasb; г) cmpsw?

9. Який з варіантів резервування пам’яті у секції .data? для змінної з типом запис є помилковим:

а) A DATA {}; б) R DATA <>; в) D DATA []; г) Жодного?

10. Яка з команд не належить до групи логічних:

а) AND; б) XOR; в) NOT; г) ABS?

11. Яка з директив не є керуючою секціями програми:

а) .stack; б) .586p; в) .const; г) .code?

12. Якого з регістрів не існує у МП Intel64/AMD64 :

а) ES; б) HS; в) FS; г) SS?

13. Який із прапорів RFLAGS є прапором перенесення:

а) CF; б) DF; в) SF; г) IF?

14. Програма для перетворення вихідного тексту (коду) програми в об’єктний файл (код) – це…

а) Рефактор; б) Компанувальник; в) Лінкер; г) Транслятор.

15. Вихідний текст програми на асемблері складається з:

а) діалогів, команд, макрокоманд і коментарів;

б) директив, команд, макрокоманд і коментарів;

в) директив, процесів, коментарів;

г) мікрокоманд, коментарів і діалогів.

16. Найпростіший тип адресації операнда в пам’яті – це…

а) базова адресація; б) індексна адресація з масштабуванням;

в) пряма адресація; г) базово-індексна адресація зі зміщенням.

17. Виберіть директиви керування секціями даних:

а) .stack, .ret; б) .link, .end; в) .data, .data?; г) .start, .const.

18. Основне призначення директиви DT – це підтримка …

а) шістнадцяткової арифметики; б) двійково-десяткової арифметики;

в) двійкової арифметики; г) десяткової арифметики.

19. Виберіть сегментні регістри:

а) CS, ES, SS, FS, GS; б) CS, DS, AS, SS, GS;

в) EIP, CS, GS, FS, SS; г) AS, BS, CS, DS.

20. Символом роздільником полів запису в асемблері MASM є…

а) крапка; б) кома; в) знак питання; г) двокрапка.

21. Який з регістрів є вказівником на верхівку стека:

а) EBX; б) EBP; в) ESP; г) EBD?

22. Яка з наступних команд виконує арифметичне зрушення ліворуч:

а) SHR; б) SAR; в) SHL; г) SAL?

23. Команда ADD є…

а) триадресною; б) двохадресною; в) одноадресною; г) безадресною.

24. Який з регістрів не можна безпосередньо використовувати як операнд команд:

а) CS; б) AX; в) DI; г) CX?

25. Із яким регістром пов’язана команда LOOP:

а) EBP; б) ECX; в) SIL; г) EBP?

26. Яка з перелічених команд виконує логічне зрушення праворуч:

а) SAL; б) SHL; в) SAR; г) SHR?

27. Команда MUL є…

а) двохадресною; б) безадресною; в) триадресною; г) одноадресною.

28. У якому регістрі зберігається залишок від ділення на байт:

а) BL; б) BH; в) AH; г) AX?

29. Яка з команд виконує дію сума за модулем 2:

а) NOT; б) XOR; в) AND; г) OR?

30. Якщо в регістрі прапорців ZF=1, то це означає, що результат попередньої команди…

а) дорівнює 0; б) є від’ємним; в) дорівнює 1; г) є додатнім.

31. Якщо в регістрі прапорців SF=1, то це означає, що результат попередньої команди є…

а) нуль; б) додатнім; в) від’ємним; г) одиниця.

32. Команда ADD – це команда…

а) додавання; б) множення; в) ділення; г) логічна.

33. Що робить команда CALL:

а) страхує на випадок переповнення; б) викликає підпрограму;

в) говорить о закінчені підпрограми; г) викликає рядок?

34. Яка з команд копіює значення з регістра AL:

а) STOSB; б) STOSW; в) STOSD; г) STOSQ?

35. Регістр акумулятор – це…

а) RBX; б) EAX; в) EDI; г) SIL.

36. У якому коді мікропроцесор виконує арифметичні операції:

а) у прямому; б) у коді Шеннона; в) у додатковому; г) к коді Фур’є?

37. Асемблер – це мова…

а) візуального програмування; б) формальна;

в) низького рівня; г) для підтримки .Net і C#.

38. Параметри передаються через стек у прямому порядку (зліва направо), стек вирівнює підпрограма в конвенції виклику…

а) PASCAL; б) C (CDECL); в) STDCALL; г) FASTCALL.

39. Параметри передаються через стек у зворотному порядку (справа наліво), стек вирівнює код, що викликає підпрограму в конвенції виклику…

а) PASCAL; б) C (CDECL); в) STDCALL; г) FASTCALL.

40. Параметри передаються через стек у зворотному порядку (справа наліво), стек вирівнює підпрограма в конвенції виклику…

а) PASCAL; б) C (CDECL); в) STDCALL; г) FASTCALL.

41. Частина параметрів передається в регістрах, а інші, за наявності таких, розміщуються в стеку в зворотному порядку (справа наліво), стек вирівнює підпрограма в одній з реалізацій конвенції виклику…

а) PASCAL; б) C (CDECL); в) STDCALL; г) FASTCALL.

42. Чим відрізняються конвенції виклику C (CDECL) і STDCALL:

а) нічим не відрізняються;

б) порядком передавання параметрів у стеку;

в) місцем розташуванням параметрів;

г) тим, хто займається вирівнюванням стека?

43. Чим відрізняються конвенції виклику PASCAL і STDCALL:

а) нічим не відрізняються;

б) порядком передавання параметрів у стеку;

в) місцем розташуванням параметрів;

г) тим, хто займається вирівнюванням стека?

44. Що спільне в реалізації FASTCALL (REGISTER) Embarcadero C++ Builder і Microsoft Visual C++ для 32-х розрядних версій Windows NT:

а) регістри, використовувані для параметрів;

б) порядок передавання параметрів в стеку;

в) усе спільне – вони ідентичні;

г) підпрограма займається вирівнюванням стеку?

45. Які регістри використовуються Microsoft Visual C++ для передавання параметрів у реалізації FASTCALL для 32-розрядних версій Windows NT:

а) EBX і ECX; б) ECX і EDX; в) EAX, EBX і ECX; г) EAX, ECX і EDX?

46. Які регістри використовуються Embarcadero C++ Builder для передавання параметрів у реалізації FASTCALL (REGISTER) для 32-розрядних версій Windows NT:

а) EBX і ECX; б) ECX і EDX; в) EAX, EBX і ECX; г) EAX, ECX і EDX?

47. У якому регістрі передається перший (зліва направо) параметр у реалізації FASTCALL у Microsoft Visual C++ для 32-розрядних версій Windows NT:

а) EAX; б) EBX; в) ECX; г) EDX?

48. У якому регістрі передається другий (зліва направо) параметр у реалізації FASTCALL у Microsoft Visual C++ для 32-розрядних версій Windows NT:

а) EAX; б) EBX; в) ECX; г) EDX?

49. У якому регістрі передається перший (зліва направо) параметр у реалізації FASTCALL (REGISTER) в Embarcadero C++ Builder для 32-розрядних версій Windows NT:

а) EAX; б) EBX; в) ECX; г) EDX?

50. У якому регістрі передається другий (зліва направо) параметр у реалізації FASTCALL (REGISTER) в Embarcadero C++ Builder для 32-розрядних версій Windows NT:

а) EAX; б) EBX; в) ECX; г) EDX?

51. Де передається третій (зліва направо) параметр у реалізації FASTCALL у Microsoft Visual C++ для 32-розрядних версій Windows NT:

а) у регістрі EAX; б) у регістрі EDX; в) у регістрі ECX; г) у стеку?

52. Де передається третій (зліва направо) параметр у реалізації FASTCALL (REGISTER) в Embarcadero C++ Builder для 32-розрядних версій Windows NT:

а) у регістрі EAX; б) у регістрі EDX; в) у регістрі ECX; г) у стеку?

53. Яку конвенцію за замовчуванням використовує компілятор Microsoft Visual C++, якщо вона не зазначена при описі прототипів використовуваних функцій бібліотеки у вихідному тексті:

а) __cdecl; б) __stdcall; в) __fastcall; г) __thiscall?

54. Із використанням DLL замість статичних бібліотек…

а) програмні файли займають менше місця на диску;

б) програмні модулі займають менше місця в пам’яті;

в) швидше здійснюється завантаження програм на виконання;

г) легше виконується перенесення програм з комп’ютера на комп’ютер.

55. Перевагою використання пізнього зв’язування з DLL, порівняно з раннім, є те, що…

а) програма за необхідності може вивантажити DLL, звільнивши використовувану пам’ять;

б) неможливо завантажити на виконання програму за відсутності DLL з якою вона була зв’язана;

в) DLL не завантажується у адресний простір додатка, що зменшує розмір використовуваної пам’яті;

г) на момент запуску програми всі необхідні DLL виявляються завантаженими автоматично.

56. Перевагою використання раннього зв’язування з DLL, порівняно з пізнім, є те, що…

а) програма за необхідності може вивантажити DLL, звільнивши використовувану пам’ять;

б) неможливо завантажити для виконання програму за відсутності DLL, з якою вона була зв’язана;

в) DLL не завантажується у адресний простір додатка, що зменшує розмір використовуваної пам’яті;

г) на момент запуску програми всі необхідні DLL виявляються завантаженими автоматично.

57. Із використанням SafeDllSearchMode у Windows NT пошук DLL починається…

а) із системного каталогу Windows; б) із каталогу встановлення Windows;

в) із поточного каталогу додатка; г) із поточного каталогу користувача.

58. Із використанням SafeDllSearchMode у Windows NT пошук DLL завершується…

а) у каталозі встановлення Windows; б) у каталогах, зазначених у PATH;

в) у поточному каталозі додаткаа; г) у поточному каталозі користувача.

59. Якщо SafeDllSearchMode у Windows NT не задіяний, то пошук DLL починається…

а) із системного каталогу Windows; б) вз каталогів, зазначених у PATH;

в) із поточного каталогу додатка; г) вз поточного каталогу користувача.

60. Якщо SafeDllSearchMode у Windows NT не задіяний, то пошук DLL завершується…

а) у каталозі встановлення Windows; б) у каталогах, зазначених в PATH;

в) у 16-розрядному системному каталозі; г) у поточному каталозі користувача.

61. Із використанням SafeDllSearchMode у Windows NT пошук DLL…

а) у системному каталозі Windows виконується після пошуку в каталозі встановлення Windows;

б) у поточному каталозі користувача виконується відразу після пошуку в поточному каталозі додатка;

в) у системному каталозі Windows виконується відразу після пошуку в поточному каталозі додатка;

г) у каталогах, зазначених у PATH виконується перед пошуком у системному каталозі Windows.

62. Із використанням SafeDllSearchMode у Windows NT пошук DLL…

а) у каталозі встановлення Windows виконується перед пошуком у системному каталозі Windows;

б) у поточному каталозі користувача виконується після пошуку в каталозі встановлення Windows;

в) у поточному каталозі користувача виконується після пошуку в каталогах, зазначених у PATH;

г) у каталогах, зазначених у PATH, виконується перед пошуком у каталозі встановлення Windows.

63. Якщо SafeDllSearchMode у Windows NT не задіяний, то пошук DLL…

а) у системному каталозі Windows виконується після пошуку в каталозі встановлення Windows;

б) у поточному каталозі користувача виконується відразу після пошуку в поточному каталозі додатка;

в) у системному каталозі Windows виконується відразу після пошуку в поточному каталозі додатка;

г) у каталогах, зазначених у PATH, виконується перед пошуком у системному каталозі Windows.

64. Якщо SafeDllSearchMode у Windows NT не задіяний, то пошук DLL…

а) у системному каталозі Windows виконується після пошуку в каталозі встановлення Windows;

б) у поточному каталозі користувача виконується після пошуку в каталозі встановлення Windows;

в) у системному каталозі Windows виконується після пошуку в поточному каталозі користувача;

г) у каталогах, зазначених у PATH, виконується перед пошуком у поточному каталозі користувача.

65. Зазначити неправильне твердження щодо функціонування DLL у Windows NT:

а) DLL може експортувати тільки функції і не може експортувати дані та/або ресурси;

б) DLL завжди виконується у контексті певного потоку певного процесу;

в) у раз використання DLL двома та більше додатками її секції коду і даних є поділюваними;

г) DLL має вхідну точку DllMain, однак не може виконуватися самостійно.

66. Зазначити неправильне твердження щодо функціонування DLL у Windows NT:

а) секції DLL відображаються у адресний простір кожного додатка, що її використовує;

б) DLL може мати як секцію коду, так і секцію даних/ресурсів, але не має власного стека;

в) код DLL завжди виконується у контексті первинного потоку процесу, що її використовує;

г) для секцій DLL операційна система використовує механізм копіювання під час запису.

67. Якщо параметр fdwReason функції DllMain дорівнює DLL_PROCESS_ATTACH, а параметр lpvReserved функції DllMain НЕ дорівнює нулю, то DLL…

а) завантажується в адресний простір внаслідок статичного (раннього) зв'язування;

б) завантажується в адресний простір процесу внаслідок динамічного (пізнього) зв'язування;

в) вивантажується з адресного простору процесу внаслідок завершення процесу;

г) вивантажується з адресного простору процесу внаслідок виклику функції FreeLibrary.

68. Якщо параметр fdwReason функції DllMain дорівнює DLL_PROCESS_ATTACH, а параметр lpvReserved функції DllMain дорівнює нулю, то DLL…

а) завантажується в адресний простір процесу внаслідок статичного (раннього) зв'язування;

б) завантажується в адресний простір процесу внаслідок динамічного (пізнього) зв’язування;

в) вивантажується з адресного простору процесу внаслідок завершення процесу;

г) вивантажується з адресного простору процесу внаслідок виклику функції FreeLibrary.

69. Якщо параметр fdwReason функції DllMain дорівнює DLL_PROCESS_DETACH, а параметр lpvReserved функції DllMain НЕ дорівнює нулю, то DLL…

а) завантажується в адресний простір процесу внаслідок статичного (раннього) зв’язування;

б) завантажується в адресний простір процесу внаслідок динамічного (пізнього) зв’язування;

в) вивантажується з адресного простору процесу внаслідок завершення процесу;

г) вивантажується з адресного простору процесу внаслідок виклику функції FreeLibrary.

70. Якщо параметр fdwReason функції DllMain дорівнює DLL_PROCESS_DETACH, а параметр lpvReserved функції DllMain дорівнює нулю, то DLL…

а) завантажується в адресний простір процесу внаслідок статичного (раннього) зв’язування;

б) завантажується в адресний простір процесу внаслідок динамічного (пізнього) зв’язування;

в) вивантажується з адресного простору процесу внаслідок завершення процесу;

г) вивантажується з адресного простору процесу внаслідок виклику функції FreeLibrary.

71. Зазначити неправильне твердження щодо функції DllMain:

а) через параметр hinstDLL функції DllMain система повідомляє DLL її власний ідентифікатор;

б) DllMain викликається системою за певних подій і її призначення полягає в ініціалізації DLL;

в) функція DllMain викликається системою лише один раз – у момент зв'язування з DLL;

г) якщо fdwReason не дорівнює DLL_PROCESS_ATTACH, то код повернення DllMain ігнорується.

72. Який тип даних не підтримує математичний співпроцесор:

а) 16-бітне ціле число; б) коротке 32-бітне дійсне число;

в) неупаковане дійсне BCD-число; г) негативний нуль?

73. Який тип даних не підтримує математичний співпроцесор:

а) 48-бітне ціле число; б) довге 64-бітне дійсне число;

в) упаковане ціле BCD-число; г) позитивна і негативна нескінченність?

74. Який тип даних не підтримує математичний співпроцесор:

а) 64-бітне довге ціле число; б) розширене 80-бітне дійсне число;

в) упаковане 128-бітне дійсне число; г) сигнальне нечисло SNAN?

75. Який тип даних не підтримує математичний співпроцесор:

а) розширене 80-бітне довге ціле число; б) розширене 80-бітне дійсне число;

в) денормалізоване дійсне число; г) тихе нечисло QNAN?

76. Який тип даних не підтримує математичний співпроцесор:

а) 64-бітне довге ціле число; б) розширене 80-бітне дійсне число;

в) упаковане 80-бітне ціле BCD-число; г) тихе несигнальне число QNSAN?

77. Регістр тегів співпроцесора (TR, Tag Register) надає інформацію про…

а) вміст регістрів стека співпроцесора;

б) положення верхівки стека співпроцесора;

в) поточний стан співпроцесора;

г) замасковані виключення співпроцесора.

78. Положення верхівки стека математичного співпроцесора…

а) у будь-який момент часу жорстко зв'язане з регістром R0;

б) визначається значенням поля TOP регістру слова стану SWR;

в) визначається значенням поля PC регістру керування CWR;

г) визначається значенням відповідних бітів регістру тегів TR.

79. Виникнення різних виключних ситуацій математичного співпроцесора фіксується…

а) установленням відповідної виключенню комбінації прапорців C0, C1, C2, C3 в SWR;

б) установленням в одиничне значення прапорців SF і C1 в SWR;

в) установленням в одиничне значення одного з шести молодших бітів SWR;

г) установленням в одиничне значення одного з шести молодших бітів CWR.

80. Зазначити неправильне твердження щодо виключень співпроцесора:

а) для маскування виключень використовуються молодші шість бітів CWR;

б) виключення арифметики з плаваючою комою (#MF) генерується тільки за встановленого біта ES;

в) замасковані виключення обробляються самим математичним співпроцесором;

г) виникнення замаскованого виключення встановлює одиничне значення біта ES у регістрі SWR.

81. У разі виникнення виключення арифметики з плаваючою комою в регістрі OR зберігається…

а) 11 молодших біт машинного коду команди, яка виконувалася на момент виникнення виключення;

б) адреса команди, яка виконувалась на момент виникнення виключення;

в) адреса операнда у пам’яті команди, яка виконувалась на момент виникнення виключення;

г) значення верхівки стека співпроцесора на момент виникнення виключення.

82. У разі виникнення виключення арифметики з плаваючою комою в регістрі DPR зберігається…

а) 11 молодших біт машинного коду команди, яка виконувалася на момент виникнення виключення;

б) адреса команди, яка виконувалася на момент виникнення виключення;

в) адреса операнда у пам’яті команди, яка виконувалася на момент виникнення виключення;

г) значення верхівки стека співпроцесора на момент виникнення виключення.

83. У разі виникнення виключення арифметики з плаваючою комою в регістрі IPR зберігається:

а) 11 молодших біт машинного коду команди, яка виконувалася на момент виникнення виключення;

б) адреса команди, яка виконувалася на момент виникнення виключення;

в) адреса операнда у пам’яті команди, яка виконувалася на момент виникнення виключення;

г) значення верхівки стека співпроцесора на момент виникнення виключення.

84. Обробка математичним співпроцесором замаскованих виключень полягає у тому, що він…

а) звільняє верхівку стека і встановлює в одиничне значення відповідне поле регістру тегів;

б) встановлює в одиницю біти PM, UM, OM, ZM, DM, IM регістру CWR;

в) залежно від ситуації, розміщує в регістрах стека одне зі спеціальних числових значень;

г) залежно від ситуації, установлює певну комбінацію бітів C0–C3 і виконує реініціалізацію.

85. Поле RC регістра CWR математичного співпроцесора…

а) керує режимом округлення;

б) визначає точність обчислень;

в) визначає напрямок зростання стека;

г) містить код останнього виключення.

86. Поле PC регістра CWR математичного співпроцесора…

а) керує режимом округлення; б) визначає напрямок зростання стека;

в) визначає точність обчислень; г) містить код останнього виключення.

87. Різниця команд FINIT і FNINIT полягає у тому, що…

а) перед виконанням FINIT маскуються усі виключення, а перед виконанням FNINIT – ні;

б) перед виконанням FINIT перевіряється наявність немаскованого виключення, а перед FNINIT – ні;

в) у результаті виконання FNINIT маскуються усі виключення, а в результаті виконання FINIT – ні;

г) виконання FINIT завершується генерацією виключення, а виконання FINIT – ні.

88. Команда FSTSW <операнд>…

а) копіює значення регістра слова стану співпроцесора в <операнд>, що може бути словом у пам’яті або регістром AX;

б) завантажує значення <операнд> в регістр керування співпроцесора;

в) зберігає у буфер пам’яті розміром не менше, ніж 28 байтів за адресою, що визначає <операнд> вміст регістрів CWR, SWR, TAG, IPR, OP, DPR;

г) установлює в нульове значення біти B, SF, ES, PE, UE, OE, ZE, DE регістра слова стану співпроцесора.

89. Команда FLDCW <операнд>…

а) копіює значення регістра слова стану співпроцесора в <операнд>, що може бути словом в пам’яті або регістром AX;

б) завантажує значення <операнд> у регістр керування співпроцесора;

в) зберігає у буфер пам’яті розміром не менше ніж 28 байтів за адресою, що визначає <операнд> вміст регістрів CWR, SWR, TAG, IPR, OP, DPR;

г) зберігає у буфер пам’яті за адресою, що визначається <операнд> вміст регістрів CWR, SWR, TAG, IPR, OP, DPR, R0-R7 і після цього виконує реініціалізацію математичного співпроцесора.

90. Команда FLDENV <операнд>…

а) копіює значення регістра слова стану співпроцесора в <операнд>, що може бути словом у пам’яті або регістром AX;

б) завантажує значення <операнд> у регістр керування співпроцесора;

в) завантажує в регістри CWR, SWR, TAG, IPR, OP, DPR вміст буфера пам’яті за адресою, що визначає <операнд>;

г) завантажує в регістри CWR, SWR, TAG, IPR, OP, DPR, R0-R7 вміст буфера пам’яті за адресою, що визначає <операнд>.

91. Команда FRSTOR <операнд>…

а) установлює в 11 значення поля TR, відповідне регістру даних співпроцесора, зв’язаному з <операнд>, не змінюючи вміст самого регістра і поля TOP в SWR;

б) завантажує значення <операнд> у регістр керування співпроцесора;

в) завантажує в регістри CWR, SWR, TAG, IPR, OP, DPR вміст буфера пам’яті за адресою, що визначає <операнд>;

г) завантажує в регістри CWR, SWR, TAG, IPR, OP, DPR, R0-R7 вміст буфера пам’яті за адресою, що визначає <операнд>.

СПИСОК ЛІТЕРАТУРИ

1. Рисований О. М. Системне програмування: підручник для студентів напрямку «Комп'ют ерна інженерія» вищих навчальних закладів у 2-х томах. Том 1. Видання четверте: виправлено та доповнено – Харків: «Слово», 2015. 576 с.
2. Рисований О. М. Системне програмування: підручник для студентів напрямку «Комп'ютерна інженерія» вищих навчальних закладів у 2-х томах. Том 2. Видання четверте: виправлено та доповнено – Харків: «Слово», 2015. 378 с.
3. Alexey Lyashko. Follow Mastering Assembly Programming: From instruction set to kernel module with Intel processor 1st ed. Packt Publishing, 2017. 290 p.
4. Daniel Kusswurm. Modern X86 Assembly Language Programming: Covers x86 64-bit, AVX, AVX2 and AVX-512 1st ed. Apress, 2018. 625 p.
5. Jeff Duntemann. Follow Assembly Language Step-by-Step 3rd ed. Wiley, 2011. 656 p.
6. Jo Van Hoey. Beginning x64 Assembly Programming, From Novice to AVX Professional. Apress, 2019. 432 p.
7. Irvine Kip. Assembly Language for x86 Processors 8th ed. Pearson, 2019. 880 p.
8. Randall Hyde. The Art Of 64-bit Assembly. No Starch Press, 2021. 1032 p.
9. Ray Seyfarth. Introduction to 64 Bit Windows Assembly Language Programming 8th ed. CreateSpace Independent Publishing Platform, 2017. 288 p.
10. Sherwyn Allibang. Assembly Language: Simple, Short, and Straightforward Way of Learning Assembly Programming. Independently published, 2020. 160 p.
11. Інтернет ресурс: Intel® 64 and IA-32 Architectures Software Developer Manuals https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.
12. Інтернет ресурс: Microsoft Macro Assembler reference https://learn.microsoft.com/en-us/cpp/assembler/masm/microsoft-macro-assembler-

reference?view=msvc-170.

13. Інтернет ресурс: The MASM Forum http://masm32.com/board/index.php.

Навчальне видання

Зілінський Юрій Володимирович
Перекрест Андрій Леонідович
Юдіна Анна Леонідівна

СИСТЕМНЕ ПРОГРАМУВАННЯ. ПРОГРАМУВАННЯ НА АСЕМБЛЕРІ.

Навчальний посібник

Підп. до др. 12.07.2023.Формат 60x84 1/16. Папір тип.
Друк цифровий. Ум. друк. арк. 7,3. Наклад 300 прим. Зам. № 03/23

Редакційно-видавничий відділ
Кременчуцького національного університету
імені Михайла Остроградського
вул. Університетська, 20, м. Кременчук, 39600
Реєстраційне свідоцтво серії ДК № 4837 від 22.01.2015