Міністерство освіти і науки України
Національний технічний університет України
Київський політехнічний інститут імені Ігоря Сікорського

**П. Ю. Катін**

# Архітектура комп’ютера

## Лабораторний практикум

Електронне мережеве видання

*Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського
як навчальний посібник для здобувачів ступеня бакалавра за спеціальностями
121 «Інженерія програмного забезпечення», 126 «Інформаційні системи та технології»,
151 «Автоматизація та комп’ютерно-інтегровані технології»*

Київ
КПІ ім. Ігоря Сікорського
2021

УДК 004.2(076.5)
К29

*Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського*
*(Протокол № 08 від 24.06.2021)*

**Рецензент**
*О. М. Долголенко,* канд. техн. наук, доц., ст. наук. співроб.,
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

*М. М. Ткач*, канд. техн. наук, доц.,
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

**Катін П. Ю.**

К29 Архітектура комп’ютера : лаб. практикум [Електронний ресурс]: навч. посібн. для студ. спец. 121 «Інженерія програмного забезпечення» / П. Ю. Катін. – Електронні текстові дані (1 файл: 2,2 Мбайт). – Київ : КПІ ім. Ігоря Сікорського, 2021. – 123 с.

Уміщено завдання на лабораторні роботи, приклади їх розв’язання та порядок оформлення результатів лабораторних робіт. Надано прототипи (шаблони) коду для використання у лабораторних роботах та додатковий інформаційний матеріал.

Для студентів всіх форм навчання, які навчаються за спеціальністю 121 «Інженерія програмного забезпечення» факультету інформатики та обчислювальної техніки КПІ ім. Ігоря Сікорського. Може бути використаний для спеціальностей 126 «Інформаційні системи та технології» та 151 «Автоматизація та комп'ютерно-інтегровані технології».

**УДК 004.2(076.5)**

# ЗМІСТ

# ВСТУП

Нині найбільш розповсюдженими термінами для архітектури сучасних мікропроцесорів персональних комп'ютерів є AMD64, Intel® 64 (EM64T) або x86-64 [1–5]. Історично склалося так, що перший 64-розрядний мікропроцесор для персонального комп'ютера (ПК) був розроблений компанією AMD і представлений у 2000 році. Саме тому термін AMD64 є доволі розповсюдженим для 64-розрядної архітектури мікропроцесорів ПК. Зрозуміло, що архітектура AMD64 та Intel®64 мають певні незначні відмінності, проте виробники максимально адаптують основні елементи архітектури для розробників операційних систем. Використання терміну AMD64 не означає певної переваги одної фірми виробника процесорів над іншим виробником. На ринку мікропроцесорів для персональних комп'ютерів (та інших подібних систем) є два основних виробника – Intel та AMD. Сучасні POSIX сумісні операційні системи, операційні системи родини Microsoft або інші типи сучасних ОС однаково працюють на ПК на основі мікропроцесорів цих двох фірм виробників.

Потрібно також нагадати, що нині активно розвивається архітектура ARM, що суттєво відрізняється від архітектури AMD64 (Intel®64 (EM64T) або x86-64). Мікропроцесори на основі ARM також використовуються для розроблення ноутбуків (або інших аналогічних систем). Не дивлячись на це, мікропроцесори архітектури AMD64 (Intel® 64 (EM64T) або x86-64) є найбільш розповсюдженими для ПК. У цьому навчальному посібнику розглянуто саме цю архітектуру і далі буде використовуватися термін AMD64 для її визначення.

Архітектура AMD64 (Intel® 64) дає можливість забезпечити 64-розрядні обчислення у персональних комп'ютерах або вбудованих пристроях і забезпечує такі потенційні характеристики: 64-розрядний простір віртуальних адрес; 64-розрядні покажчики для доступу до адресного простору; 64-розрядні регістри загального призначення для зберігання проміжних результатів і обчислення; 64-розрядна підтримка обчислень з цілими числами; адресацію потенційно до 1 ТБ адресного простору. Звичайно, що практичні характеристики залежать від конкретної реалізації процесора і можуть бути значно менші перерахованих вище.

На основі архітектури AMD64 (Intel®64) виготовляються багатоядерні мікропроцесори для настільних ПК, ноутбуків, мікрокомп'ютерів або інших аналогічних систем. Нині загальнодоступними є топові мікропроцесори архітектури AMD64 (Intel®64) різних виробників, що мають до 36 обчислювальних ядер і до 72 потоків. Вони можуть адресувати більше ніж 200 Гігабайт оперативної пам'яті.

Мікропроцесори архітектури AMD64 (Intel® 64) мають два основних режими роботи [1–2]:

– *режим Long Mode*, що призначений для 64-розрядних ОС і дає можливість використати всі переваги 64-розрядного режиму роботи мікропроцесора;

– *режим Legacy Mode* (режим забезпечення сумісності), що призначений для підтримки 32-розрядних ОС 64-розрядного мікропроцесора.

У режимі Legacy Mode є три базових режими роботи мікропроцесора, це [1]:

– *захищений режим роботи (Protected mode*), призначений для роботи 32-розрядних ОС;

– *віртуальний 8086 режим роботи (Virtual-8086 mode*), призначений для сумісності 32-розрядних ОС і 16-розрядного програмного забезпечення, практично нині не актуальний;

– *реальний режим роботи (Real mode)*, призначений для запуску 16-розрядного ПЗ, у цьому режимі процесор знаходиться до завантаження основної ОС.

Цей курс ЛР призначений для вивчення архітектури мікропроцесорів ПК у реальному режимі роботи (Real mode) у варіанті Legacy Mode. У курсі лабораторних робіт також розглядаються основи захищеного режиму роботи (Protected mode) у варіанті Legacy Mode.

У навчальному посібнику є стандартний підхід із навчання програмуванню (Асемблер), коли на перших етапах (лабораторних роботах) виконується відносно складне завдання, що дає можливість зацікавити студентів до подальшого навчання у реальному режимі роботи (Real mode).

Фінальним завданням є програмний додаток, що частково моделює роботу BIOS сучасної комп'ютерної системи у реальному режимі роботи (Real mode).

# ЛАБОРАТОРНА РОБОТА 1
# ТЕХНОЛОГІЯ РОЗРОБЛЕННЯ ПРОГРАМ У REAL MODE

Мета лабораторної роботи полягає у набутті знань, умінь та навичок з технології розроблення ПЗ на Асемблері для мікропроцесора архітектури AMD64 (Intel® 64) у реальному режимі. Під час використання Асемблера як мови програмування застосовуються знання архітектури мікропроцесора персонального комп'ютера.

*Перелік обладнання для виконання курсу лабораторних робіт 1–8.* Для підготовки ЛР № 1–8 може бути використаний будь-який сучасний ПК на основі мікропроцесора AMD64 (Intel® 64). Для розробки, налагодження і тестування мікропроцесора у реальному режимі роботи (Real mode) у варіанті Legacy Mode потрібно застосовувати програми віртуалізації. Приклади ЛР протестовані на DosBox. Можуть бути використані інші варіанти віртуальних машин.

На першому етапі навчання обраний Асемблер TASM, після перших 3-х лабораторних робіт може бути обраний інший тип Асемблера для цієї архітектури. Перша лабораторна робота виконується під керівництвом викладача.

## *Завдання для лабораторної роботи 1*

1. Виконати повний технологічний цикл створення програми на Асемблері TASM для мікропроцесора архітектури AMD64 (Intel® 64) у реальному режимі роботи. Результати ЛР протестувати на віртуальній машині DosBox. Можуть бути використані інші варіанти віртуальних машин за вибором студента.

2. Доопрацювати шаблон вихідного коду програми для виведення на консоль прізвищ всіх студентів робочої бригади. Виконати цю вправу також з використанням процедур.

3. З використанням Turbo Debugger провести покрокове виконання програми, дослідити всі архітектурні елементи, що використані у програмі Асемблера. Зробити висновки щодо зв'язку вихідного коду й архітектури.

## *Програма проведення експерименту лабораторної роботи 1*

1. Створення вихідного коду. Користуючись одним з текстових редакторів, створити файл вихідного коду лабораторної роботи hello.asm.

Зберегти його у робочому каталозі, що містить програмну інфраструктуру Асемблера. Доробити вихідний код відповідно до поданого завдання.

2. В операційній системі Windows 10 або у сучасній POSIX операційній системі використовувати віртуальну машину, наприклад DosBox. Запустити цю програму віртуалізації і, використовуючи консоль DosBox, перейти до робочого каталогу.

3. Далі за допомогою програми TASM.EXE провести асемблювання вихідного коду. Для асемблювання вихідного коду у командному рядку виконується така команда:

*tasm.exe /l /zihello.asm.*

Докладний опис команди подано далі у теоретичних відомостях до ЛР 1.

4. Компонування або лінкування. У командному рядку виконується така команда:

*tlink.exe/v hello.obj.*

Докладний опис команди подано далі у теоретичних відомостях до ЛР 1.

5. У разі появи помилок на етапі асемблювання внести зміни до вихідного коду і повторити етапи 3, 4.

6. Після успішного виконання п. 3, 4 запустити програму на виконання і на трасування. Для трасування програми необхідно виконати у командному рядку у робочому каталозі таку команду:

*td.exe hello.exe.*

7. Після покрокового виконання програми зробити скриншоти екранів і створити звіт, додаючи до звіту скриншоти екранів. Зробити висновки щодо потреби знання архітектури під час розробки програми на Асемблері.

## *Теоретичні відомості для ЛР 1*

Мікропроцесори архітектури AMD64 (Intel® 64) мають два основних режими роботи [1–2]:

– *режим Long Mode*, що призначений для 64-розрядних ОС і дає можливість використати всі переваги 64-розрядного режиму роботи мікропроцесора;

– *режим Legacy Mode* (режим забезпечення сумісності), що призначений для підтримки 32-розрядних ОС 64-розрядного мікропроцесора. Режим

сумісності підтримує двійкову сумісність з наявними 16- і 32-розрядними версіями.

У режимі Legacy Mode є три базових режими роботи мікропроцесора [1]:

– *захищений режим роботи (Protected mode*) призначений для роботи 32-розрядних ОС, підтримує 16- та 32-розрядні програми, дає можливість відокремити адресний простір кожного процесу, забезпечувати рівень привілеїв для ядра ОС і програм користувача, може адресувати до 4 Гб пам'яті;

– *віртуальний 8086 режим роботи (Virtual-8086 mode*) призначений для сумісності 32-розрядних ОС і 16-розрядного програмного забезпечення, практично нині не актуальний;

– *реальний режим роботи (Real mode)* – у цьому режимі процесор перебуває до завантаження основної ОС, не підтримує захист пам'яті і розподіл рівнів доступу, може надавати доступ до адресації 1 Мб оперативної пам'яті.

Для першого завдання пропонується розробити просту програму на Асемблері, що буде виводити на екран повідомлення, зупинятися й очікувати натискання клавіші користувачем. На цьому її функціональність завершується. Вона створена з використанням Асемблера TASM для архітектури I8086. Нині це рішення актуальне для *реального режиму роботи (Real mode)* до завантаження операційної системи і переходу у захищений режим.

Розглянемо технологічні етапи створення програми.

*Етап 1*. Створення вихідного коду. Приклад вихідного коду показаний у фрагменті коду, що надано далі. Для розроблення необхідно, користуючись одним з текстових редакторів, створити файл hello.asm. Скопіювати фрагмент коду до файлу hello.asm і зберегти його у робочому каталозі. Переробити вихідний код програми відповідно до наданого завдання.

*Етап 2*. Асемблювання вихідного коду здійснюється з використанням Асемблера TASM. Для роботи з цим Асемблером у 16-розрядному режимі в операційній системі Windows 10 або у сучасній POSIX операційній системі доцільно використовувати віртуальну машину, наприклад DosBox. У процесі роботи потрібно запустити цю програму віртуалізації і, використовуючи консоль DosBox, підключити робочий каталог, перейти до робочого каталогу. Далі за допомогою програми TASM.EXE проводимо асемблювання вихідного

коду, тобто переводимо файл вихідного коду hello.asm у файл hello.obj. Для асемблювання вихідного коду у командному рядку виконується така команда:

*tasm.exe /l /zihello.asm.*

Прапорець */l* створює у робочому каталозі файл лістингу *hello.lst*, що містить адреси, машинні коди, текст, коментарі. Це інформація для налагодження.

Прапорець */x* створює файл карти пам'яті. Наприклад під час виконання команди *tlink /x hello.obj* створюється файл карти пам'яті *hello.map* (Linked Address Map).

Прапорець /zi додає до файлу hello.obj коментарі, назви змінних тощо. Це необхідно для спрощення процесу налагодження. У випадку відсутності помилок у hello.asm у робочому каталозі з'являється об'єктний файл *hello.obj*.

*Eman 3*. Лінкування. На цьому етапі завершується визначення адресних посилань й об'єднання, якщо потрібно, декількох об'єктних файлів і бібліотек. У командному рядку виконується така команда:

*tlink.exe/v hello.obj.*

Прапорець */v* додає до файлу *hello.exe* інформацію для спрощення процесу налагодження (коментарі, назви змінних тощо). Якщо *eman 3* є успішним, створюється файл *hello.exe.* Його можна запустити, виконуючи команду *hello.exe* й отримати у консолі *Helloworld.*

Крім того, для проведення експерименту необхідно проводити покрокове виконання програми, контролювати зміст змінних, регістрів і дампу пам'яті.

Для цього використовується Turbo Debugger, що надається у комплекті з Асемблером TASM. Для трасування програми необхідно виконати у командному рядку у робочому каталозі таку команду:

*td.exe hello.exe.*

Фрагмент коду, що може бути взятий за основу, подано нижче.

```
TITLE ЛР_1_1
;---------------------------------------------------------------------------
;ЛР  № 1-1
;---------------------------------------------------------------------------
;
; Завдання:
; ВНЗ:              КПІ ім. Ігоря Сікорського
; Факультет:    ФІОТ
```

```
; Курс:    1          1
; Група:       __ __ __
;-------------------------------------------------------------------------
; Автор:         __ __ __
; Дата:          __/__/__
;--------------------------------
; I. ЗАГОЛОВОК ПРОГРАМИ
IDEAL   ; Директива – тип Асемблера tasm
MODEL small  ; Директива – тип моделі пам’яті
STACK 256     ; Директива – розмір стека
; II. МАКРОСИ
; III. ПОЧАТОК СЕГМЕНТА ДАНИХ
DATASEG
exCode db 0
message db "Hello world!",10,13,'$'; Рядок символів для виведення на екран
; VI. ПОЧАТОК СЕГМЕНТА КОДУ
CODESEG
Start:
;--------------------------------1. Ініціалізація DS і ES-
mov ax,@data; @data ідентифікатор, що створюються директивою model
mov ds, ax      ; Завантаження початку сегмента даних у регістр ds
mov es, ax      ; Завантаження початку сегмента даних у регістр es
;--------------------------------;2. Операція виведення на консоль
; Пересилання адреси рядка символів message в регістр dx
mov dx, offset message
; Завантаження числа 09h до регістру ah
; (Функція DOS 9h – команда виводу на консоль рядка)
mov ah,09h
int 21h  ; Виклик функції DOS 9h
;--------------------------------3. Операція зупинки програми, очікування натискання
клавіш-----
mov ah,01h
; Завантаження числа 01h до регістру ah
; (Функція DOS 1h – команда очікування натискання клавіші...)
int 21h  ; Виклик функції DOS 21h
; Завантаження числа 4ch до регістру ah
; (Функція DOS 4ch – вихід з програми)
;--------------------------------4. Вихід з програми
mov ah,4ch
mov al,[exCode]        ; отримання коду виходу
int 21h  ; виклик функції DOS 4ch
end Start
```

Будь-яка програма на Асемблері складається з окремих частин. Ці частини програми, у багатьох випадках, розміщуються в окремих ділянках пам’яті, що називаються сегментами. Механізми адресації, що пов’язані з цим визначенням будуть більш детально розглянуті у подальшому курсі лабораторних робіт.

Як можна побачити з рис. 1.1, налагоджувач (TD) відображає вікно коду, дамп пам’яті, регістри, вікностека і прапорці процесора. Ліворуч вікна коду і дампу пам’яті знаходяться логічні адреси для кожного рядка програми або рядка змісту пам’яті, наприклад 1955:0003.

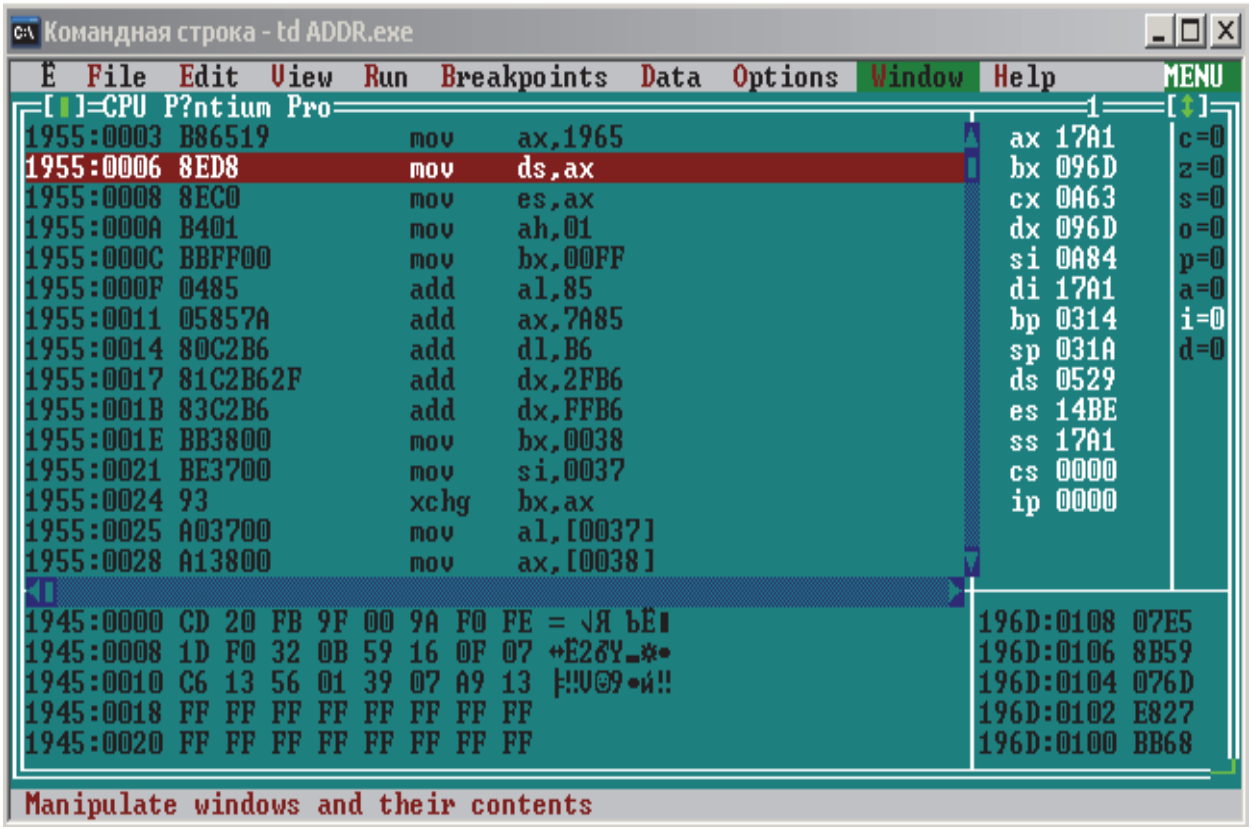



*Рис. 1.1*. Вікно програми Turbo Debugger

Деяким сегментам програми у реальному режимі відповідають фрагменти вихідного коду на Асемблері. Для шаблону вихідного коду ЛР 1 такими фрагментами є сегмент даних і сегмент коду. Сегмент стека не відображається у вихідному коді. Розглянемо складові частини вихідного коду більш докладно. Перша частина – це заголовок програми, що показано нижче.

```
; I. ЗАГОЛОВОК ПРОГРАМИ
IDEAL                          ; Директива – тип Асемблера tasm
MODELsmall                     ; Директива – тип моделі пам’яті
STACK256                  ; Директива – розмір стека
```

Перший рядок заголовку – директива, що визначає тип діалекту Асемблера TASM. Асемблер TASM підтримує два діалекти: TASM і MASM. Директива IDEAL у нашому прикладі визначає діалект TASM.

Другий рядок заголовку – директива MODEL. Вона визначає тип моделі пам’яті, у цьому прикладі – це тип small. Це означає, що передбачається використання 3-х сегментів у програмі, а саме: сегмента коду, даних і стека. Третя директива заголовку програми визначає розмір стека, що є програмно-апаратним механізмом цієї архітектури і його опису присвячена ЛР 4.

Друга частина програми – це сегмент даних. Опишемо більш докладніше сегмент даних прикладу коду 1.1. Перший рядок (DATASEG) визначає початок сегмента даних у вихідному коді.

```
; III. ПОЧАТОК СЕГМЕНТА ДАНИХ
DATASEG
exCode db 0
message db "Hello world!",10,13,'$'; Рядок символів для виведення на екран
```

Сегмент даних призначений для збереження змінних, масивів й інших даних програми. У подальших рядках прикладу сегменту даних визначаються глобальні змінні. Наприклад, змінна exCode, що має тип (розмір) 1 байт та початкове значення 0. Друга змінна є рядком символів, що будуть виводитися на консоль. Фактично message – це покажчик на нульовий елемент масиву символів.

Наступна частина вихідного коду – сегмент коду. Він визначається таким чином:

```
; VI. ПОЧАТОК СЕГМЕНТА КОДУ
CODESEG
Start:
```

Сегмент коду містить вихідний код програми, саме ту її складову, що реалізує алгоритм. Таким чином програму на Асемблері в цій моделі пам’яті можна умовно розподілити на три сегменти, два з яких явно описані у файлі вихідного коду.

Для розуміння роботи програми Асемблера необхідно уявляти, як виконується адресація даних, які регістри процесора й у який спосіб можуть використовуватися під час виконання інструкцій. Для цього необхідно розглянути базову програмну модель мікропроцесорів архітектури AMD64 (Intel® 64) у реальному режимі роботи. Вона значно збігається з програмною моделлю архітектури I8086. До програмної моделі зазвичай входять:

– 8 регістрів загального призначення, що служать для зберігання даних і адрес;

– регістри сегментів зберігають адреси початку сегментів програми;

– регістр прапорців FLAGS, що дозволяє керувати виконанням програми і зберігає поточний стан процесора;

– регістр-покажчик IP процесора містить команду, що буде виконуватися у наступному кроці програми;

– система команд (інструкцій) процесора, звичайно є індивідуальною для кожної окремої архітектури процесора;

– режими адресації даних у командах процесора, звичайно є індивідуальною для кожної окремої архітектури процесора, хоча містить деякі однакові риси.

Розглянемо більш докладно, що таке регістр. Регістр – це ділянка пам'яті, що вбудована до мікропроцесора. Набір регістрів мають всі архітектури мікропроцесорів і мікроконтролерів. Кожний регістр має власну особливу назву, що доступна в Асемблері і найбільш швидкий доступ для запису і читання. В архітектурі процесорів AMD64 (Intel® 64) у реальному режимі підтримується апаратно набір регістрів архітектури процесора Intel 8086 та інші додаткові регістри.

Регістри архітектури процесора Intel 8086 мають розмір 2 байти або одне машинне слово (16 біт). За кожним з регістрів закріплені власні імена, з використанням яких регістри адресуються в Асемблері.

Регістри розподілені на групи, першою з яких є група регістрів загального призначення. Чотири регістри загального призначення AX, BX, CX, DX мають можливість адресувати окремо старший і молодший байт та мають для них окремі назви. Кожний з цих регістрів може використовуватися у програмі для збереження результатів різних операцій. Проте кожний цей регістр має свої особливості. Розглянемо особливості цих регістрів більш докладно:

AX (AH, AL) – акумулятор, що призначений для всіх операцій введення/виведення, більш ніж інші регістри загального призначення спеціалізований для виконання арифметичних операцій, деякі арифметичні операції виконуються виключно з його використанням;

BX (BH, BL) – базовий регістр, рівноцінний іншим регістрам загального призначення, має особливість, що характерна тільки для нього, він може зберігати адресу покажчика на масив під час базової адресації;

CX (CH, CL) – регістр-лічильник, рівноцінний іншим регістрам загального призначення, має особливість, яка характерна тільки для нього, – забезпечує автоматичний декремент змінної під час циклічних операцій;

DX (DH, DL) – регістр даних, рівноцінний іншим регістрам загального призначення, має особливість, яка характерна тільки для нього, – забезпечує введення і виведення інформації на зовнішні пристрої (операції IN, OUT). В архітектурі AMD64 (Intel® 64) також доступні 32- і 64-розрядні регістри, що показані на рис. 1.2 [1–2]. Набір регістрів для ЛР 1 виділений зеленим кольором на рис. 1.2.

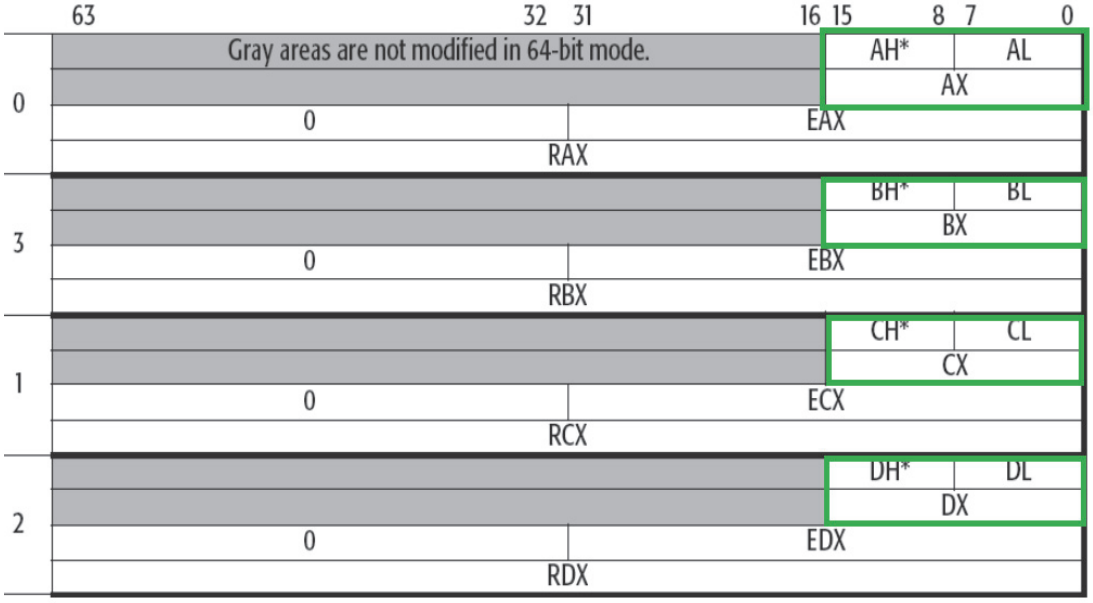



*Рис. 1.2*. Набір регістрів загального призначення у AMD64 (Intel® 64) [1]

На рис. 1.2 також показані назви для 32- і 64-розрядних регістрів.

До регістрів загального призначення також належать регістри-покажчики й індексні регістри. Вони можуть бути використані в арифметичних і логічних операціях. На відміну від вищеописаних вони не можуть окремо адресувати молодший і старший байт слова. Основне призначення регістрів-покажчиків – це забезпечення доступу до інформації у сегменті стека, а саме:

– регістр SP покажчик на стек дає можливість адресувати вершину стека, разом з регістром SS визначають адресу операнду у вершині стеку;

– регістр BP покажчик бази стека дає можливість адресувати будь-який операнд, що знаходиться у стеку, довільним чином. Отже, три регістри SP, SS, BP повністю керують стеком і не бажано їх використовувати з іншою метою.

В архітектурі AMD64 (Intel® 64) також доступні 32- і 64-розрядні регістри, що показані на рис. 1.3 [1–2]. Набір регістрів актуальних для ЛР 1 виділений зеленим кольором на рис. 1.3. Розрядність також показана на рис. 1.3.

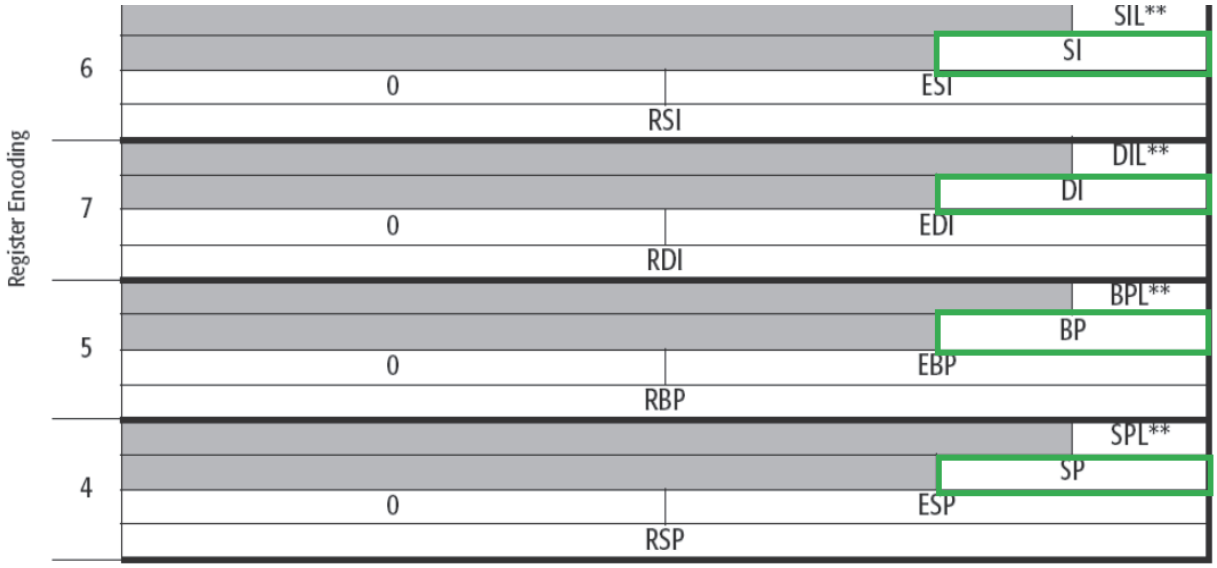



*Рис. 1.3*. Набір покажчиків й індексних регістрів у AMD64 (Intel® 64) [1]

Основне призначення індексних регістрів – виконання операції пересилання символьних рядків, а саме: регістр SI індекс джерела рядка символів, регістр DI індекс призначення рядка символів. Ці регістри також використовуються в індексній адресації. У програмах для налагодження прийнято позначати зміст регістрів з використанням цифр системи числення з основою 16.

Другою групою регістрів процесорів архітектури AMD64 (Intel® 64) у реальному режимі є сегментні регістри (рис. 1.4). Вони аналогічні регістрам архітектури процесора Intel 8086, є 16-розрядними для архітектури AMD64 (Intel® 64). В архітектурі процесорів Intel 8086 всього 4 таких регістри, що показані на рис. 1.4 зеленим кольором. В архітектурі AMD64 (Intel® 64) кількість сегментних регістрів більше ніж у Intel 8086.

Нумерація розрядів починається з 0 і закінчується 15. Сегментні регістри не призначені для виконання математичних, логічних та інших операцій. Їх

основне призначення це забезпечення адресації сегментів стека, коду, даних і додаткових сегментів. Кожний з сегментних регістрів може забезпечити адресацію 64 кілобайт ОЗП. Вони не можуть бути адресовані прямо, доступ до них здійснюється через регістри загального призначення. Опишемо їх більш докладно.

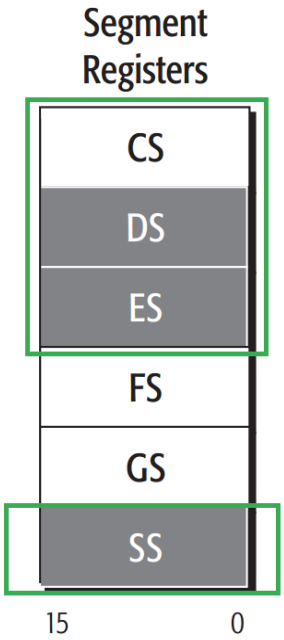



*Рис. 1.4*. Набір сегментних регістрів AMD64 (Intel® 64) [1]

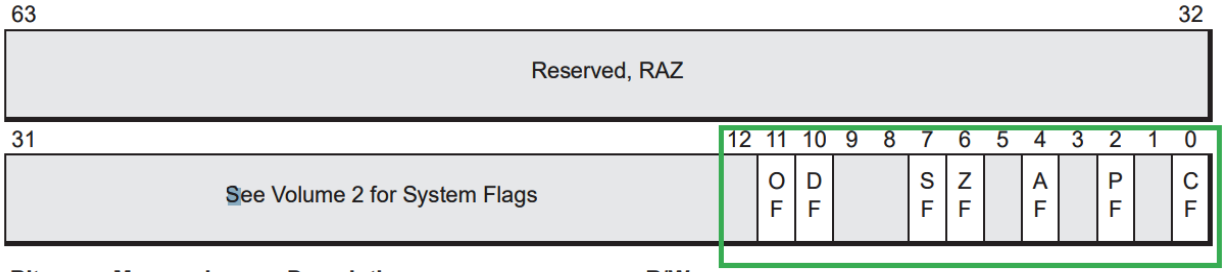
Регістр CS – визначає початок сегмента коду, він містить початкову адресу сегмента коду, зміст цього сегмента, помножений на 16 із додаванням величини зміщення у регістрі IP, визначає адресу поточної команди, що буде виконуватися. До регістра покажчика IP у розробника немає програмного доступу.

Регістр DS – визначає початок даних, він містить початкову адресу сегмента даних, у нашому прикладі вимагає ініціалізації й дає можливість адресувати будь-яку змінну, об’єкт або символьний рядок, що знаходиться у сегменті даних.

Регістр SS – визначає початок стека, він містить початкову адресу сегмента стека, зміст цього сегмента, помножений на 16 із додаванням величини зміщення у регістрі SP, визначає адресу слова у вершині стека.

Регістр ES – додатковий регістр, може містити тимчасові значення адрес сегментів у різних способах адресації, потребує ініціалізації.

Регістр прапорців в архітектурі AMD64 (Intel® 64) показано на рис. 1.5. Біти з 0–15 показані на рис. 1.5 зеленим кольором, належать до регістру FLAGS і сумісні з архітектурою I8086. Саме ці біти будуть використовуватися у ЛР 1–8. Вони зберігають або відображають поточний стан процесора під час обчислень, призначені для управління обчисленнями і системами.




| Bits | Mnemonic | Description | R/W |
|---|---|---|---|
| 11 | OF | Overflow Flag | R/W |
| 10 | DF | Direction Flag | R/W |
| 7 | SF | Sign Flag | R/W |
| 6 | ZF | Zero Flag | R/W |
| 4 | AF | Auxiliary Carry Flag | R/W |
| 2 | PF | Parity Flag | R/W |
| 0 | CF | Carry Flag | R/W |

*Рис. 1.5*. Регістр прапорців AMD64 (Intel® 64) [1]

Перерахуємо прапорці регістру FLAGS та їх призначення: *OF* переповнення – встановлюється в одиницю у разі переповнення старшого біта операнду; *DF* – визначає правий або лівий напрямок пересилання строкових рядків у пам'яті; *IF* встановлений в одиницю, коли дозволені зовнішні переривання; *TF* встановлений в одиницю у покровому режимі під час налагодження програми; *SF* – містить знак результату під час арифметичних операцій; *ZF* – ознака результату арифметичної операції, якщо результат дорівнює 0, встановлюється в 1; *AF* – зовнішній перенос, використовується для спеціалізованих арифметичних операцій; *PF* – контроль парності встановлюється в одиницю у разі парності результату у молодшій частині слова; *CF* – містить ознаку переносу зі старшого біту після арифметичних операцій, або останній біт у разі циклічних зсувів.

Як можна побачити з шаблону вихідного коду у програмі використовуються переривання DOS. Опишемо більш докладно ці переривання.

**Переривання DOS 06h**: записати символ на консоль без перевірки на Ctrl-Break

Вхідні параметри: AH = 06h

DL = ASCII-код символа

Вихідні параметри: AL = код записаного символа

**Переривання DOS 09h**: вивід рядка символів

Вхідні параметри: AH = 09h

DS:DX = адреса рядка, що закінчується символом $(24h)

Вихідні параметри: AL = код крайнього символа

Наведемо приклад переривання DOS, що застосований у наведеному прикладі вихідного коду програми 1.1 і частково розкриємо його роботу. Фрагмент показано далі.

```
;----------------------------------3. Операція зупинки програми, очікування натискання клавіш---
        mov ah,01h
                        ; Завантаження числа 01h до регістру ah
                        ; (Функція DOS 1h – команда очікування
        натискання клавіші...)int21h        ; Виклик функції DOS1h
```

У ділянці коду, що показано вище, здійснюється операція затримки програми й очікування до натискання клавіші клавіатури користувачем для її продовження. У цьому прикладі до регістру AH заноситься номер переривання DOS. У нашому випадку до AH треба занести 01 (команда mov ah, 01h). Інших вхідних параметрів переривання не потрібно. Після підготовки відразу викликається переривання DOS int 21h. Після цього робота програми зупиняється і програма очікує натискання на клавішу користувачем, після натискання програма продовжує роботу. Взагалі робота переривання DOS int 21h номер 01 дещо складніша, докладно її можна вивчити у [3–5].

Крім переривань DOS можуть бути використані переривання BIOS. Наведемо декілька прикладів переривань BIOS для виведення на консоль.

**Переривання** BIOS **10h (**00h**)**:Встановити

відеорежим Вхідні параметри: AH = 00h

AL = номер режиму у молодших 7 бітах

Вихідні параметри:

**Переривання** BIOS **10h (**02h**)**: Управління положенням курсору. Вхідні параметри: AH = 02h. Управління положенням курсору

BH = Номер сторінки

DH = рядок

DL = стовпчик

Вихідні параметри:

**Переривання** BIOS**10h (**AH = 13h**)**: Виведення рядка із заданими параметрами. Вхідні параметри: AH = 13h

AL = режим виведення, біт 0 курсор до кінця, біт 1 наявність атрибутів, кожний символ містить код ASCII (1 байт) і атрибут (2 байти)

CX = довжина рядка (тільки кількість символів)

BL = атрибут для всього рядка, якщо рядок містить символи

DH, DL = рядок і стовпчик, координати початку виведення ES:BP = адреса рядка, що виводиться у пам'яті.

Процедура Асемблера – це аналог функції у процедурних високорівневих мовах програмування (мова С) або статичного методу в ООП мовах (C++, C#, Java). Для розроблення програми з використанням процедур користувача наведемо вихідний код, що містить процедури Асемблера. У вихідному коді 1–3 наведений приклад використання процедур. У цьому прикладі виведення на консоль відбувається через процедури. У вихідному коді використані дві процедури, що виконують виведення інформації на консоль з використанням переривань DOS.

```
TITLE ЛР_1_3
;-----------------------------------------------------------------------
;ЛР  № 1-3
;-----------------------------------------------------------------------
;
; Завдання:
; ВНЗ:              КПІ ім. Ігоря Сікорського
; Факультет:    ФІОТ
; Курс:    1
; Група:       __ __ __
;-----------------------------------------------------------------------
```

```
; Автор:        __ __ __
; Дата:         __/__/__
;--------------------------------
;
; I. ЗАГОЛОВОК ПРОГРАМИ
IDEAL ; Директива - тип Асемблера tasm
MODEL small ; Директива - тип моделі пам'яті
STACK 512 ; Директива - розмір стека в байтах
;--------------------
MACRO M_Init
        mov ax, @data ; ax &lt;- @data
        mov ds, ax ; ds &lt;- ax
        mov es, ax ; es &lt;- ax
        xor si, si
ENDM M_Init
;--------------------
MACRO M_Exit
        mov ah,4ch
        mov al,[exCode]         ; отримання коду виходу
        int 21h ; виклик функції DOS 4ch
ENDM  M_Exit
;--------------------
DATASEG
exCode    db 0
message   db "Hello world!",10,13,'$' ; Рядок символів для виведення на екран
message1  db "Hello world! From proc1",10,13,'$' ; Рядок символів для виведення на екран
message2  db "Hello world! From proc2",10,13,'$' ; Рядок символів для виведення на екран
;--------------------
CODESEG
        PROC main
        M_Init

                ;       2.      DOS
                ; Пересилання адреси рядка символів message в регістр dx
                mov dx, offset message
                ; Завантаження числа 09h до регістру ah
                ; (Функція DOS 9h – команда виводу на консоль рядка)
                mov ah,09h
                int 21h ; Виклик функції DOS 9h
                ;       3.      From proc  HelloWorld1
                call HelloWorld1
                ;       4.      From proc  HelloWorld2
                mov dx, offset message2
                call HelloWorld2
;--------------------------------3. Операція зупинки програми, очікування натискання клавіш-----
mov ah,01h
int 21h

        M_Exit                  ; макрос
ENDP main

;-------------------------------------------------------------------------
PROC HelloWorld1
; Test for proc
; Parametr list:      <- no
; Output   list:      <- no
```

```
;
;-------------------------------------------------------------------------
mov dx, offset message1
                    mov ah,09h
                    int 21h ; Виклик функції DOS 9h
                    ret
ENDP HelloWorld1

          ;-------------------------------------------------------------------------
PROC HelloWorld2
; Test for proc in
; Parametr list:      <- dx has offset for string
; Output   list:      <- no
;
;-------------------------------------------------------------------------
                    mov ah,09h
                    int 21h ; Виклик функції DOS 9h
                    ret
ENDP HelloWorld2

END main
```

Для кращого розуміння функціональності, вхідних параметрів процедури і вихідних параметрів процедури вона має містити відповідні коментарі. Використання процедур дає можливість зробити вихідний код більш читабельним.

Таким чином показано, що мікропроцесори архітектури AMD64 (Intel® 64) мають два основних режими роботи [1–2]: режим Long Mode і режим Legacy Mode.

У режимі Legacy Mode є три базових режими роботи мікропроцесора, це [1]: захищений режим роботи (Protected mode), віртуальний 8086 режим роботи (Virtual-8086 mode), реальний режим роботи (Real mode). Крайній режим роботи мікропроцесора призначений для роботи 16-розрядного ПЗ, у цьому режимі процесор перебуває до завантаження основної ОС, не підтримує захист пам'яті та розподіл рівнів доступу, може надавати доступ до адресації 1 Мб оперативної пам'яті.

Для виконання ЛР 1 у реальному режимі роботи (Real mode) потрібно використовувати віртуальну машину, наприклад DosBox.

### *Перелік запитань для підготовки до лабораторної роботи*

1. Опишіть принципи мікропрограмного керування.

2. Опишіть основи двійкової системи числення, системи числення з основою 16, побітові логічні операції, види кодування в обчислювальних машинах.

3. Опишіть принципи побудови машини Фон Неймана.

4. Опишіть структурну схему типової комп'ютерної системи.

5. Опишіть загальну організацію пам'яті комп'ютерної системи, види пам'яті.

6. Розкрийте термін «Архітектура системи команд».

7. Розкрийте термін «Програмна модель».

8. Опишіть периферійні пристрої комп'ютерної системи.

9. Розкрийте етапи створення програми з використанням Асемблера tasm або іншого Асемблера, що вивчається.

10. Розкрийте основну відмінність високорівневих мов програмування і Асемблера у контексті зв'язку з архітектурою комп'ютера.

11. Опишіть технологію створення комп'ютерної програми на Асемблері.

12. Як архітектурні елементи комп'ютерної системи зв'язані з кодом Асемблера?

13. Опишіть технологію асемблювання вихідного коду, які процеси відбуваються і які файли утворюються.

14. Напишіть прапорці команди tasm для асемблювання вихідного коду. Які можливості вони надають?

15. Що відбувається на етапі асемблювання вихідного коду. Які команди викликаються (на прикладі TASM), які їх параметри?

16. Що відбувається на етапі компонування програми (TLINK). Які команди використовуються у командному рядку і які їх параметри?

17. Яку команду треба набрати для виклику Turbo Debugger (TD)? Що відображається на робочому вікні TD, як дослідити дамп пам'яті?

18. Опишіть спрощену структуру системної плати комп'ютерів і вкажіть призначення основних елементів схеми.

# ЛАБОРАТОРНА РОБОТА 2
# ПРЯМИЙ ДОСТУП ДО ВІДЕОПАМ'ЯТІ У РЕАЛЬНОМУ РЕЖИМІ

Мета лабораторної роботи полягає у набутті знань, умінь і навичок з розроблення ПЗ на Асемблері для управління відеопам'яттю у реальному режимі роботи архітектури AMD64 (Intel® 64).

## *Завдання для лабораторної роботи 2*

1. Вивести до відеопам'яті архітектури AMD64 (Intel® 64) у реальному режимі інформацію так, щоб на консолі утворився прямокутник розміром 20 знаків по горизонталі й 10 знаків по вертикалі. Колір напису подано у табл. 2.1, відповідно до варіанта. Кольори прямокутника і координати верхнього лівого кута прямокутника екрану визначені у табл. 2.1, відповідно до варіантів.

2. Прямокутник можна реалізувати програмно у такі способи:

– користуючись прикладом вихідного коду, що показаний далі;

– з використанням циклічних конструкцій для запису з сегмента даних до відеопам'яті, можна використовувати процедури для читабельності коду.

*Таблиця 2.1*

**Таблиця до завдання 1**

| Варіанти | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| Координата x | 2 | 40 | 2 | 40 | 2 | 40 | 30 | 50 |
| Координата y | 2 | 2 | 10 | 10 | 15 | 15 | 30 | 50 |
| Кольори прямокутника | Синій | Зелений | Бірюза | Червоний | Білий | Блакитний | Жовтий | Салатов. |
| Кольори надпису | Зелений | Синій | Червоний | Бірюза | Блакитний | Білий | Салатов. | Жовтий |

## *Програма проведення експерименту лабораторної роботи 2*

1. Створення вихідного коду. Користуючись одним з текстових редакторів, створити файл вихідного коду лабораторної роботи l2_gr1.asm.

Зберегти його у робочому каталозі, що містить програмну інфраструктуру Асемблера.

Доробити вихідний код відповідно до поданого завдання.

2. В операційній системі Windows 10 або у сучасній POSIX операційній системі використовувати віртуальну машину, наприклад DosBox. Запустити цю програму віртуалізації і, використовуючи консоль DosBox, перейти до робочого каталогу.

3. Далі за допомогою програми TASM.EXE провести асемблювання вихідного коду. Для асемблювання вихідного коду у командному рядку виконується така команда:

*tasm.exe /l /zil2_gr1.asm*

Докладний опис команди подано далі у теоретичних відомостях до ЛР 2.

4. Лінкування. У командному рядку виконується така команда: *tlink.exe/v l2_gr1.obj.*

Докладний опис команди подано далі у теоретичних відомостях до ЛР 2.

5. У разі появи помилок на етапі асемблювання або лінкування внести зміни до вихідного коду і повторити етапи 3, 4.

6. Після успішного виконання п. 3, 4 запустити програму на виконання. Після успішного виконання перевірити візуально вигляд прямокутника відповідно до завдання. Зробити скриншот екрану і внести результати до звіту ЛР.

7. Провести трасування програми. Для цього необхідно виконати у командному рядку у робочому каталозі таку команду:

*td.exe l2_gr1.exe.*

8. У процесі покрокового виконання за допомогою дослідження дампу пам’яті програми потрібно переконатися, що за адресою відеопам’яті знаходиться зміст, що відповідає зображенню на екрані. Зробити висновки щодо результатів роботи програми на Асемблері.

## *Теоретичні відомості для ЛР 2*

Необхідною умовою захисту лабораторної роботи є: навички розроблення вихідного коду, його асемблювання, лінкування; вміння користуватися Turbo Debugger, здійснювати покроковий запуск програми,

відкривати вікно дампу пам’яті, переходити на необхідну адресу; визначати адресу з використанням дампу пам’яті будь-якого байта у сегменті стека або сегменті коду; пояснити зміни у пам’яті програми під час кожного кроку, пояснити призначення кожної інструкції.

Для виконання роботи потрібний узагальнений опис адресації оперативної пам’яті у ПК на основі архітектури AMD64 (Intel® 64) у режимі Legacy Mode реального режиму роботи. В цьому режимі мікропроцесори перебувають до завантаження операційної системи і можуть адресувати тільки 1 Мбайт (1 048 576 байт) ОЗП (рис. 2.1). Повний обсяг оперативної пам’яті комп’ютер може адресувати тільки у режимі *Long Mode* процесора архітектури AMD64 (Intel® 64). Для мікропроцесорів архітектури AMD64 (Intel® 64) цей обсяг може досягати до 200 Гігабайт і більше оперативної пам’яті, залежно від класу процесора і системної плати.

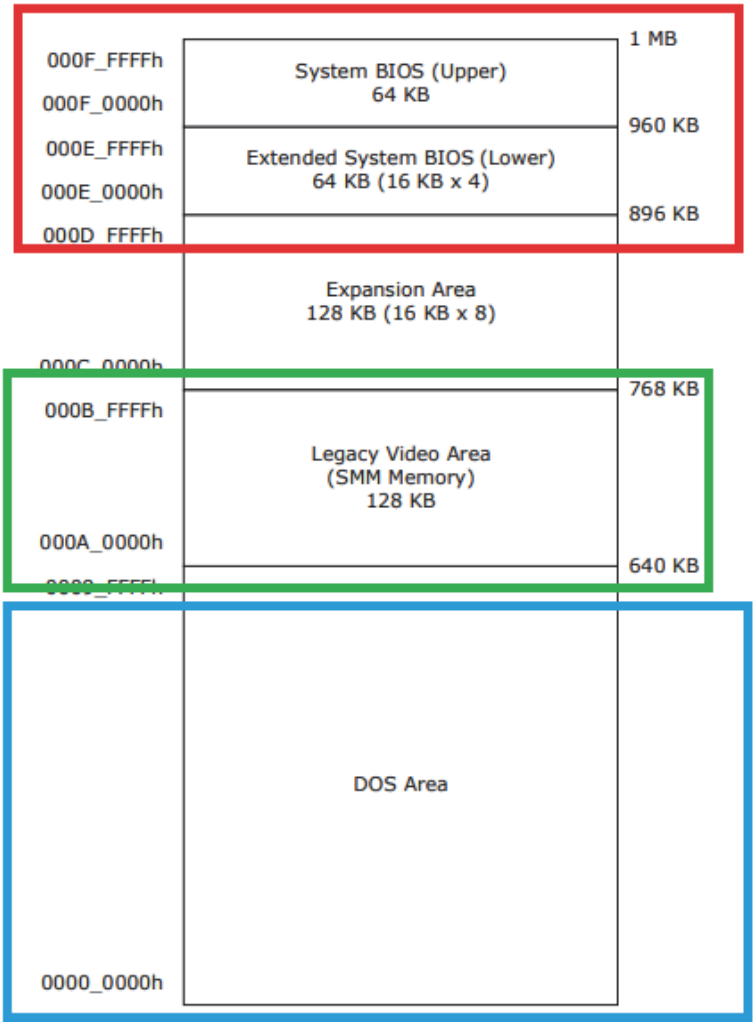



*Рис. 2.1.* Розподіл ОЗП у режимі real mode [5]

Для подальшого опису потрібно розрізняти два терміни: «адресний простір» і «оперативна пам’ять». «Адресний простір» – це набір адрес, іноді абстрактних, що може адресувати процесор. Він необов’язково відповідає реальному розміру оперативної пам’яті. Типова схема адресного простору у реальному режимі роботи для архітектури AMD64 (Intel® 64) показана на рис. 2.1. Для цього прикладу адресний простір у практичних персональних комп’ютерах у реальному режимі значно менший фактичного обсягу оперативної пам’яті. Звичайно, після завантаження операційної системи активною стає повна оперативна пам’ять персонального комп’ютера. Опишемо адресний простір у реальному режимі роботи та його використання.

Перші 640 Кбайт адресного простору фізичної пам’яті відводять під основну оперативну пам’ять, що називають стандартною або звичайною. Цьому адресному простору відповідає зміст сегментних адрес (від 0000h до 9FFFh).

На самому початку оперативної пам’яті реалізовані вектори переривань, 256 векторів по 4 байти кожний. Вони забезпечують роботу апаратної частини ПК. Стандартні вектори заповнюються автоматично, під час вивантаження інформації з BIOS на первинному етапі роботи персонального комп’ютера, до початку роботи основної операційної системи. Деякі вектори переривань розробник може додати або змінити.

За векторами переривань розміщена ділянка, що призначена для завантаження ОС й управління периферійними пристроями (клавіатура, дисплей, таймер тощо) та виконання інших функцій.

Для формування псевдографіки на екрані real mode в архітектурі AMD64 (Intel® 64) передбачена ділянка відеопам’яті, що показана на рис.2.1 (Legacy Video Area). Аналогічно реалізована ця система в архітектурі I8086. Запис певних значень до відеопам’яті автоматично призводить до появи відеозображення на екрані. У текстовому режимі відображаються відповідні символи і відбувається забарвлення екрана. За допомогою завантаження відповідної інформації до відеопам’яті у реальному режимі роботи можна формувати псевдографіку, що показується на екрані, наприклад під час переходу ПК до BIOS. Саме так формується графічний інтерфейс BIOS.

В архітектурі AMD64 (Intel® 64) це можливо тільки у реальному режимі роботи, до завантаження операційної системи і до включення у роботу драйверів відеокарти.

Відеопам’ять Legacy Video Area або I8086 починається з фізичної адреси A0000h (рис. 2.1). Аналогічно проведений розподіл відеопам’яті в операційній системі DOS в архітектурі I8086. Тому дослідження відеопам’яті архітектури I8086 у цілому еквівалентні дослідженням для архітектури AMD64 (Intel® 64) у реальному режимі.

**Текстова відеопам’ять I8086** починається з адреси B800h:0000h (для текстових режимів) і закінчується на B800h:FFFFh. Усе, що записується в цю область пам’яті, негайно пересилається в пам’ять відеоадаптера і відображається на екрані. У текстових режимах для зберігання кожного символа використовуються два байти: байт з ASCII-кодом символа і байт з його атрибутом.

Наприклад за адресою B800h:0000h має бути розміщений байт з кодом символа. Він буде відображатися у верхньому лівому кута екрану. Далі за адресою B800h:0001h має бути чисельне значення атрибуту цього символа. Атрибут визначає кольори символа, кольори екрана та інші характеристики відеозображення.

Далі за адресою B800h:0002h знаходиться код другого символа і т. д. Текстова відеопам’ять займає 32 Кбайти починаючи з адреси B8000h. Кожна сторінка займає 4 Кбайти, отже:

– сторінка B800h,

– сторінка B900h,

– сторінка BA00h тощо.

Під час включення комп’ютера видимою стає відеосторінка 0. Заміна сторінок здійснюється викликом функції 05h переривання 10h BIOS. Будь-яка інформація, що записується до відеопам’яті, відразу відображається на екрані у вигляді зафарбованого символа на певному знакомісці. Як приклад, далі подано вихідний код виведення на екран одиночного символу.

```
TITLE V cod 2.1
;-------------------------------------------------------------------------
;ЛР  № 2.1 Кодування Кирилиця Windows-1251
;-------------------------------------------------------------------------
```

```
;---------------I.ЗАГОЛОВОК ПРОГРАМИ----------------------
IDEAL
MODEL SMALL
STACK 512
;----------------II.МАКРОСИ-----------------------------------
;2.2 Складний макрос для ініціалізації
MACRO M_Init                ; Початок макросу
mov       ax, @data         ; ax <- @data
mov       ds, ax                        ; ds <- ax
mov       es, ax                        ; es <- ax
ENDM M_Init             ; Кінець макросу

;-------------------III.ПОЧАТОК СЕГМЕНТА ДАНИХ
DATASEG
exCode db 0
;---------------------VI. ПОЧАТОК СЕГМЕНТА КОДУ-----------------
CODESEG
Start:
M_Init
;---------- Variant 1. Symbols to senter 0 memory------------------
                                    ;Налаштування ES на сторінку 0 відеопам'яті
          mov AX, 0B800h            ; 1. Сегментна адреса початку відеопам'яті
          mov ES, AX                ; 2. До ES
          mov BX, 0                 ; 3. Зміщення на початок екрана
          mov SI, 2000              ; 4. Зміщення центру екрана 80*2*12+40*2
          mov DI, 3998              ; 5. Зміщення кінець екрана 80*2*25-2
          mov [word ES:BX], 3130h   ; 6. Символ на екран
;---------- INT 21 ask keyboard----------------------------
          mov AH,01h                ; 7. Останов
          int 21h
          mov [word ES:SI], 3130h   ; 8. Символ на екран
;---------- INT 21 ask keyboard----------------------------
          mov AH,01h                ; 9. Останов
          int 21h
          mov [word ES:DI], 3130h   ; 10. Символ на екран
;---------- INT 21 ask keyboard----------------------------
          mov AH,01h                ; 9. Останов
          int 21h
Exit:
mov ah,4ch
mov al,[exCode]
int 21h
end Start
```

Кожний символ займає у відеопам'яті 2 байти. Молодші (парні) байти всіх полів відводять під коди ASCII, старші (непарні) – під атрибути відповідного знака. Атрибути відповідають кольору символу та його фоні. Отже, для опису символа потрібний тип даних – слово. Розмір екрана у цьому режимі роботи

визначається 80x25 знаків. Кожному рядку екрана відповідає одновимірна матриця завдовжки у 80 знаків. Під час використання відеопам'яті перехід на наступний рядок екрана визначається не керувальними кодами ASCII, як це показано у ЛР 1, а розміщенням знака у полі сторінки пам'яті.

Залишок першого мегабайта пам'яті, передається для потреб ОС. Адреси вище 1 Мегабайта відповідають розширеній пам'яті й доступні після завантаження ОС або спеціальних драйверів.

Для виведення інформації до відеопам'яті будемо звертатися до комірки пам'яті з відомими фізичними адресами. У вихідному коді 2.1 показано приклад виведення одиночного символа на екран. Наприклад:

```
mov AX, 0B800h          ; 1. Сегментна адреса початку відеопам'яті
mov ES, AX              ; 2. До ES
mov BX, 0               ; 3. Зміщення на початок екрана

mov [word ES:BX], 3130h ; 6. Символ на екран
```

У цьому прикладі символ і атрибут *3130h* буде записаний до адреси *0B800h:0000*, тобто на початок екрана, а саме в лівий верхній кут.

Для визначення фізичної адреси потрібно знати термінологію щодо адрес. Логічна адреса складається з двох частин, наприклад ds:0000, дивись приклад коду 2.1. Першу частину адреси визначає адреса початку сегмента, що знаходиться у сегментному регістрі, наприклад ds. Для обчислення фізичної адреси це значення апаратно множиться на 16 і до нього додається зміщення у сегменті. Наприклад, нехай для певного експерименту значення у DS дорівнює 52F6.

Зміщення у сегменті дорівнює 0. Отже, його логічна адреса може бути записана DS:0000 або 52F6:0000. Проводячи обчислення можна визначити фізичну адресу, відповідно вона дорівнює 52F60.

Для запису інформації до відеопам'яті необхідно визначити фізичну адресу початку відеопам'яті і записати це значення до регістра ES. Для 0 сторінки, адреса буде дорівнювати *0B800*.

Розглянемо приклад використання TD для контролю змісту пам'яті, що показано на рис. 2.2. На ньому зображено фотографію екрана під час роботи TD при покроковому виконанні. У такий спосіб можна дослідити зміст

оперативної пам’яті сегмента даних. Аналогічно можна переключатися до потрібного сегмента.

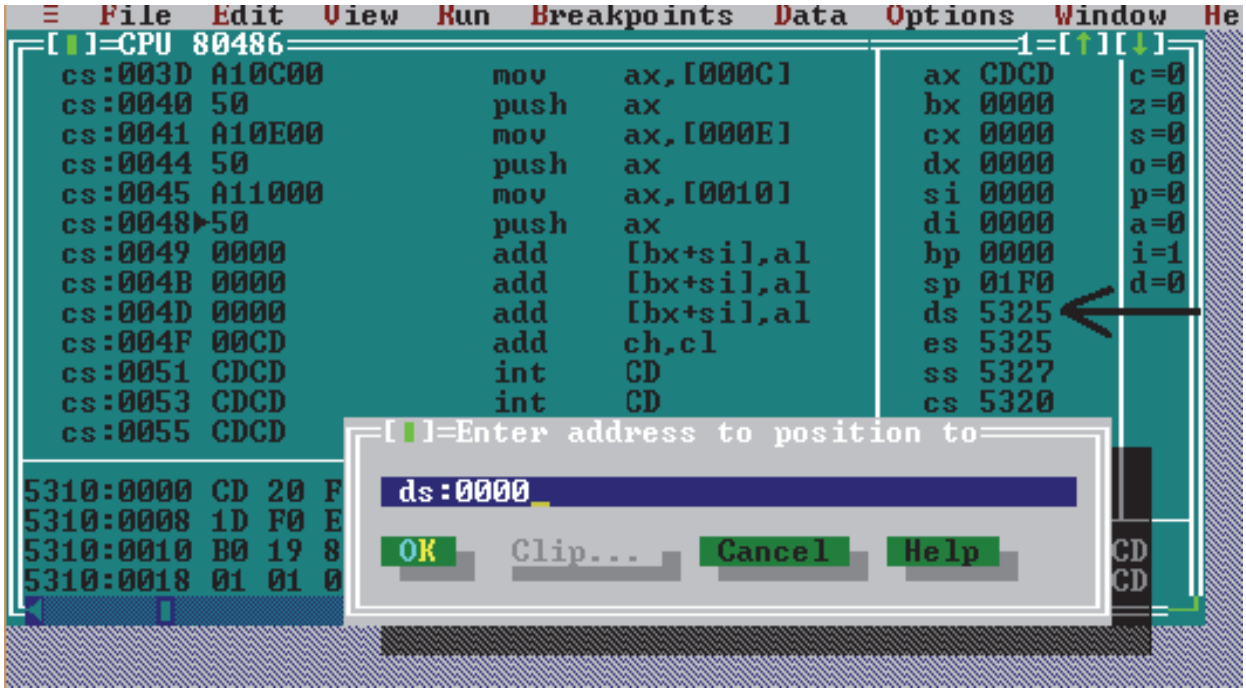



*Рис. 2.2*. Перегляд змісту сегмента даних за допомогою TD

Припустимо, що до сегмента даних записаний масив ar_ans. Переконатися у цьому можна користуючись TD, що дає можливість дослідити зміст оперативної пам’яті сегмента даних програми. Фрагмент сегмента даних показано на рис. 2.3. Усі елементи масиву розміщені в оперативній пам’яті послідовно без розривів. Нульовий елемент масиву ar_ans має логічну адресу DS:0000h, крайній байт має логічну адресу DS:0017h (рис. 2.2).


*Рис. 2.3*. Зміст початку сегмента пам’яті

Таким чином, можна визначити зміщення у сегменті для будь-якого елементу масиву або операнду. Для ar_ans[0][0], для нульового елементу масиву ar_ans зміщення, відповідно до рис. 2.3, дорівнює 0000h.

Визначимо фізичну адресу нульового елементу масиву ar_ans. Для цього зміст сегментного регістра, що, відповідно до рис. 2.2, містить число 5325h, множимо на 10h(16) і додаємо значення зміщення (ефективної

адреси). Отже, для ar_ans[0][0] фізична адреса в оперативній пам'яті дорівнює 53250h.

Один з варіантів розв'язання завдання ЛР 2 показано далі у вихідному коді 2.2. Програма реалізована у такий спосіб: до сегмента даних у вигляді масиву *rect_line* записана інформація, що містить коди символів і атрибути символів; на наступному кроці ця інформація буде переписана до відеопам'яті, 0 сторінка. Тип масиву однобайтовий.

```
TITLE ЛР_2_2
;-------------------------------------------------------------------------
;ЛР  № 2-2 АК
;-------------------------------------------------------------------------
;
; Завдання:
; ВНЗ:                КПІ ім. Ігоря Сікорського
; Факультет:    ФІОТ
; Курс:    1
; Група:       __ __ __
;-------------------------------------------------------------------------
; Автор:        __ __ __
; Дата:         __/__/__
;--------------------------------
IDEAL
MODEL small
STACK 256
MACRO M_Init ; Початок макросу
mov ax,@data ; ax <- @data
mov ds,ax ; ds <- ax
mov es,ax ; es <- ax
ENDM M_Init ; Кінець макросу

DATASEG
exCode db 0
rect_line db 'T', 012h, 'e', 012h, 's', 012h, 't', 012h, 'T', 012h
db '', 012h, 'e', 012h, 's', 012h, 't', 012h, 'M', 012h, 'e', 012h, '!', 012h ; рядок символів для виводу
; на екран, парні байти - символи, непарні - атрибути, у нашому випадку - колір фона
;символа синій, а колір самого символу зелений
db 8 dup ("", 00h, 012h)
; директива 8 dup створює 8 символів з кодом
; 00h і синім кольором, це є частиною фону прямокутника

CODESEG
Start:
M_Init
mov ax, 0B800h ; сегментна адреса текстової відеопам'яті
```

```
mov es, ax
mov bx, 0 ; у регістр BX записуємо параметр 0 (регістр BX виступає у якості лічильника)
mov di, 324 ; у регістр DI записуємо дані з регістру DX

main_loop:
mov cx, 40 ; записуємо параметр 40 у регістр CX для виводу рядка
mov si, offset rect_line ; пересилаємо адресу рядка rect_line у сегментний регістр SI
rep movsb ; записуємо у комірку з адресою ES:DI байт з комірки з адресою DS:SI,
;прапорець DF рівний нулю, тому операція пересилає дані зліва направо,
;збільшуючи значення у регістрах SI та DI на значення, яке вказане у регістрі CX
add di, 120 ; зміщення для наступної лінії
inc bx ; збільшуємо значення у регістрі BX на 1
cmp bx, 10 ; порівнюємо значення в регістрі BX з 10
jne main_loop ; якщо значенні не рівні
Exit:

mov ah, 04Ch
mov al, [exCode] ; Вхідний параметр для функції 4ch
int 21h
END Start
```

Програма реалізована через команду  movsb, що докладно описана в [1–4]. Основні етапи виконання програми можна описати у вигляді лінійного алгоритму.

Після запуску програми відбувається запис до відеопам’яті (0 сторінка) вихідного масиву і на екрані відображається потрібний малюнок у вигляді псевдографіки.

Отже, мікропроцесори на архітектурі AMD64 (Intel® 64) у режимі Legacy Mode у реальному режимі можуть адресуватися 1 Мбайт (1 048 576 байт) оперативної пам’яті. Повний обсяг оперативної пам’яті комп’ютер може адресувати тільки у режимі *Long Mode*, цей обсяг може досягати більше 200 Гігабайт після завантаження 64-розрядної ОС.

Текстова відеопам’ять у реальному режимі починається з адреси B800h:0000h (для текстових режимів) і закінчується на B800h:FFFFh. Усе, що записується в цю область пам’яті, негайно пересилається в пам’ять відеоадаптера і відображається на екрані. У текстових режимах для зберігання кожного символа використовуються два байти: байт з ASCII-кодом символа і байт з його атрибутом.

Користуючись Асемблером у режимі Legacy Mode у реальному режимі можна записувати значення до відеопам’яті для їх відеозображення.

## *Перелік запитань для підготовки до лабораторної роботи*

1. Які шини системної плати ПК ви знаєте? Опишіть їх призначення.

2. Опишіть загальну архітектуру мікропроцесора I8086. Як ця архітектура пов’язана з сучасними ПК?

3. Опишіть типи, регістри й організацію пам’яті мікропроцесора I8086.

4. На які групи можна поділити систему команд мікропроцесора 8086? Наведіть приклади команд із кожної групи.

5. Наведіть приклади виконання команд, які впливають на окремі прапорці (CF, OF, PF, SF, ZF, AF).

6. Опишіть команди передачі даних Асемблера мікропроцесора I8086 користуючись довідковою таблицею. Назвіть операції, операнди, прапорці, на які впливає команда. Наведіть приклади.

7. Опишіть арифметичні та логічні команди Асемблера мікропроцесора I8086 користуючись довідковою таблицею. Назвіть операції, операнди, прапорці, на які впливає команда. Наведіть приклади.

8. Опишіть команди Асемблера мікропроцесора I8086 для управління процесом виконання програми користуючись довідковою таблицею.

9. Опишіть команди Асемблера мікропроцесора I8086 для процедур і переривань.

10. Наведіть приклад реалізації програмного розгалуження, організації програмного циклу з використанням Асемблера.

# ЛАБОРАТОРНА РОБОТА 3
# ДОСЛІДЖЕННЯ МЕХАНІЗМІВ АДРЕСАЦІЇ

Мета лабораторної роботи полягає у вивченні механізмів адресації для архітектури AMD64 (Intel® 64) у реальному режимі роботи, набутті впевнених знань і навичок розроблення ПЗ на Асемблері, у процесі якого застосовуються знання архітектури комп'ютерів.

Метою роботи також є навчання використанню механізмів адресації архітектури AMD64 (Intel® 64) у реальному режимі роботи для розроблення ПЗ на Асемблері й отримання досконалого коду.

## *Завдання на лабораторну роботу 3*

1. Створити двовимірний масив array2Db, що описано у табл. 3.1, відповідно до варіантів. Під час створення масиву записати його до сегмента даних. Тип елементів цього масиву обрати відповідно до варіантів табл. 3.1.

2. Розробити програму, що дозволяє записати на діагональ масиву ініціали студентів групи, користуючись видами адресації, що показані у табл. 3.1. Діагональ починається з лівого верхнього кута масиву.

*Таблиця 3.1*

**Варіанти характеристик масиву і видів адресації**

| **Варіант** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |
|---|---|---|---|---|---|---|---|
| Розмірність двовимірного масиву | 16x16 | 16x20 | 16x23 | 16x25 | 16x30 | 16x32 | 16x12 |
| Тип масиву | байт | слово | байт | слово | байт | слово | байт |
| Тип адресації | базова | індексна | базово-індексна | базова | індексна | базово-індексна | базова |

## *Програма проведення експерименту лабораторної роботи 3*

1. Створення вихідного коду. Користуючись одним з текстових редакторів, створити файл вихідного коду лабораторної роботи l3_gr1.asm.

Зберегти його у робочому каталозі, що містить програмну інфраструктуру Асемблера. Записати до сегмента даних вихідний масив і масив з ініціалами студентів робочої групи.

Доробити вихідний код відповідно до поставленого завдання.

2. В операційній системі Windows 10 або у сучасній POSIX операційній використовувати віртуальну машину, наприклад DosBox. Запустити цю програму віртуалізації і, використовуючи консоль DosBox, перейти до робочого каталогу.

3. Далі за допомогою програми TASM.EXE провести асемблювання вихідного коду. Для асемблювання вихідного коду у командному рядку виконується така команда:

*tasm.exe /l /zil3_gr1.asm*

Докладний опис команди подано у теоретичних відомостях до ЛР 3.

4. Компонування або лінкування. У командному рядку виконується така команда: *tlink.exe/v l3_gr1.obj.*

Докладний опис команди подано у теоретичних відомостях до ЛР 3.

5. У разі появи помилок на етапі асемблювання або лінкування внести зміни до вихідного коду і повторити етапи 3, 4.

6. Після успішного виконання п. 3, 4 провести трасування програми. Для цього необхідно виконати у командному рядку у робочому каталозі таку команду:

*td.exe l3_gr1.exe.*

7. У процесі покрокового виконання за допомогою дослідження дампу пам’яті програми потрібно зафіксувати масив до роботи програми і після роботи програм. Це робиться за допомогою скриншотів (фотографування екранів під час роботи TD). Переконатися, що в результаті роботи програми виконані дії відповідно до завдання. Записати результати до звіту. Зробити висновки щодо результатів роботи програми на Асемблері.

## *Теоретичні відомості для ЛР 3*

Мікропроцесори архітектури AMD64 (Intel® 64) мають два основних режими роботи [1–2]: *режим Long Mode*, що призначений для 64-розрядних

ОС і *режим Legacy Mode* (режим забезпечення сумісності), що призначений для підтримки 32-розрядних ОС.

Розглянемо більш докладно набір регістрів у цих режимах. У режимі *Legacy Mode* програмно доступними є такі регістри загального призначення:

– вісім 8-бітових регістрів (AH, AL, BH, BL, CH, CL, DH, DL);

– вісім 16-розрядних регістрів (AX, BX, CX, DX, DI, SI, BP, SP);

– вісім 32-розрядних регістрів (EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP).

Програмно доступним є 16-розрядний регістр FLAGS, або 32-розрядний регістр EFLAGS. Для програмного управління використовується 16-розрядний регістр IP або 32-розрядний регістр EIP. Вони містять адресу наступної інструкції, що потрібно виконати. Узагальнено всі ці регістри показано на рис. 3.1. Зеленим кольором відокремленні регістри, що є сумісними з архітектурою I8086.

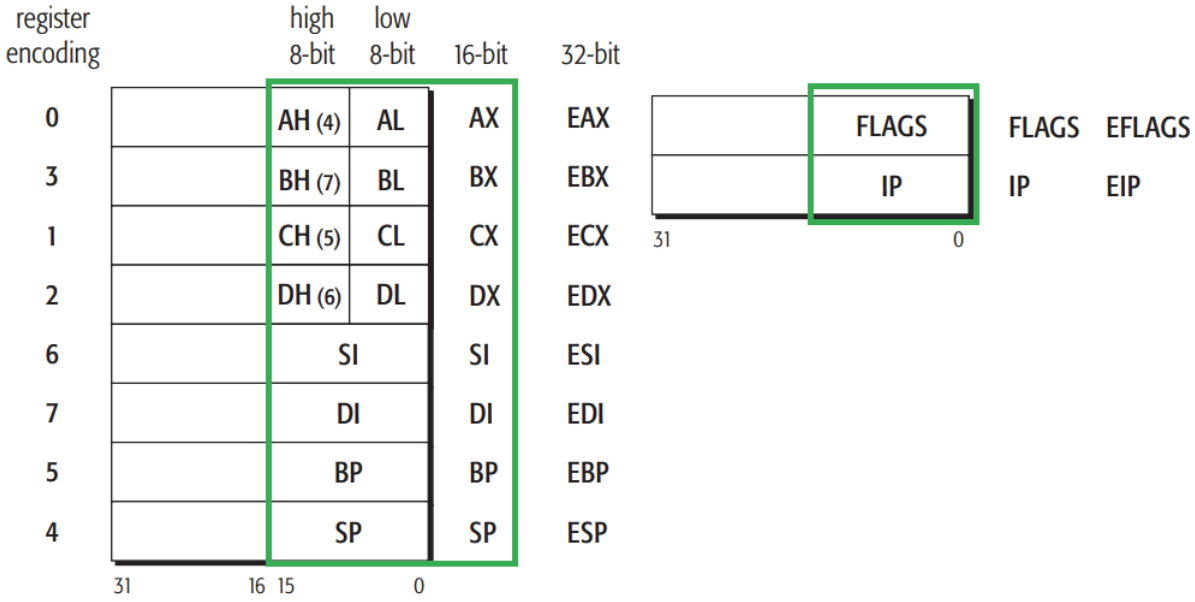



*Рис. 3.1.* Регістри у режимі Legacy Mode AMD64 (Intel® 64) [1]

У режимі Long Mode (64-розрядному режимі) до відомих регістрів загального призначення (рис. 3.1) додаються нові регістри загального призначення, що мають розмірність 64 біти. Доступними програмно залишаються також регістри з режиму сумісності *Legacy Mode* AMD64 (Intel® 64). Узагальнену схему регістрів у режимі Long Mode показано на рис. 3.2. До загального переліку регістрів належать:

– шістнадцять 8-бітових регістрів (AL, BL, CL, DL, SIL, DIL, BPL, SPL, R8B, R9B, R10B, R11B, R12B, R13B, R14B, R15B);

– чотири 8-бітові регістри (AH, BH, CH, DH);

– шістнадцять 16-розрядних регістрів (AX, BX, CX, DX, DI, SI, BP, SP, R8W, R9W, R10W, R11W, R12W,R13W, R14W, R15W);

– шістнадцять 32-розрядних регістрів (EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP, R8D, R9D, R10D, R11D,R12D, R13D, R14D, R15D);

– шістнадцять 64-розрядних регістрів (RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, R8, R9, R10, R11, R12, R13, R14, R15).

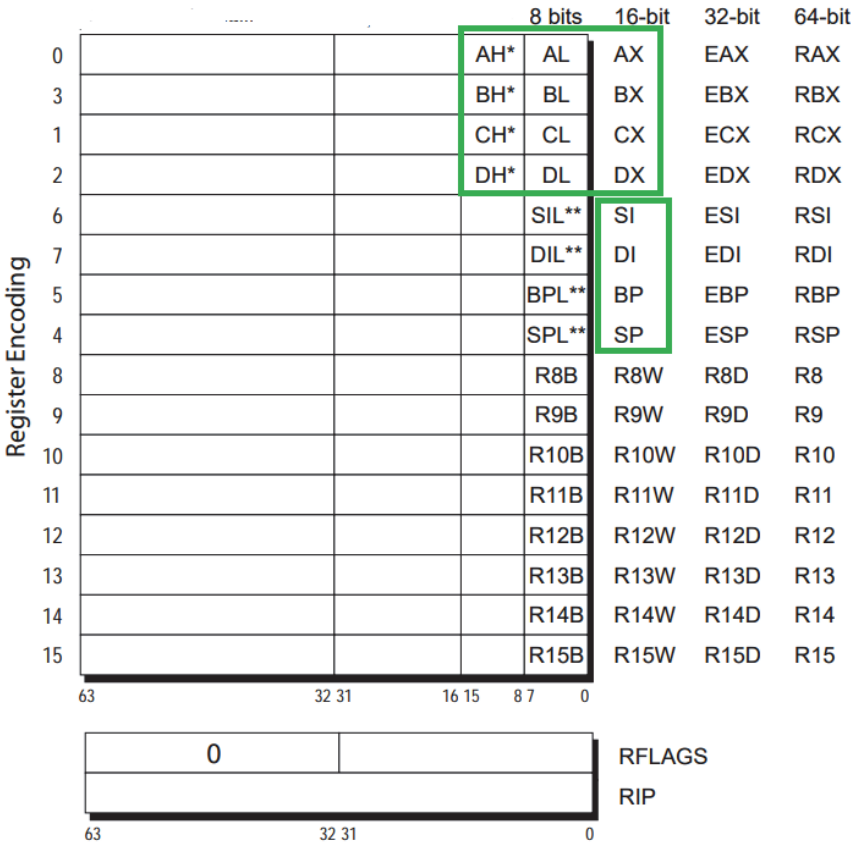



*Рис. 3.2*. Головні регістри у Long Mode (64-розрядному режимі) [1]

Зеленим кольором на рис. 3.2 показано регістри, що є сумісними з регістрами архітектури I8086.

Мікропроцесори архітектури AMD64 (Intel® 64), залежно від режимів роботи, по-різному адресують оперативну пам’ять. У режимі Legacy Mode є три базових режими роботи мікропроцесора, що визначають механізми адресації й алгоритми роботи системи ПК у процесі адресації.

*Захищений режим роботи (Protected mode*) призначений для роботи 32-розрядних ОС, підтримує 16- та 32-розрядну систему команд. У цьому режимі актуальними є сегментація, підкачка та перевірка привілеїв під час

роботи пам’яті. У дуже спрощеному вигляді схему роботи пам’яті показано на рис. 3.3. У сегментний регістр розміщується селектор, що вказує на відповідний дескриптор у таблицях дескрипторів, рівень доступу і тип таблиць дескрипторів (глобальна таблиця дескрипторів або локальна). Кожний процес у ОС за рахунок цього режиму має окремий адресний простір, а рівень ядра ОС відокремлюється від рівня користувацьких програм. Кожний процес у ОС має віртуальний адресний простір до 4 Гб оперативної пам’яті.

*Віртуальний 8086 режим роботи (Virtual-8086 mode*) призначений для сумісності 32-розрядних ОС і 16-розрядного програмного забезпечення, практично не актуальний нині.

*Реальний режим роботи (Real mode)* призначений для 16-розрядного ПЗ, у цьому режимі процесор перебуває до завантаження основної ОС, не підтримує захист пам’яті та розподіл рівнів доступу, може надавати доступ до адресації 1 Мб оперативної пам’яті. У дуже спрощеному вигляді схему роботи показано на рис. 3.3. У сегментному регістрі знаходиться адреса початку сегмента. Під час роботи механізму адресації апаратно відбувається множення цього значення на 16, або зсув ліворуч на 4 розряди. Таким чином утворюється 20-бітна адреса. До цього значення додається зміщення, що знаходиться у 16-розрядному регістрі.

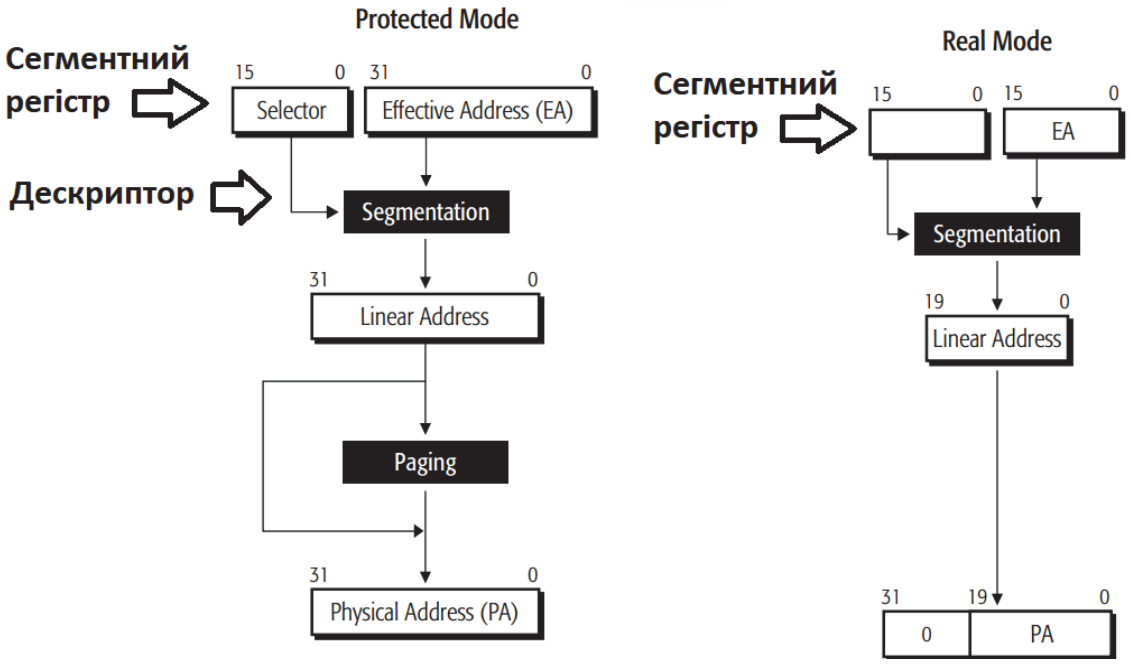



*Рис. 3.3*. Схема роботи пам’яті у Legacy Mode архітектури AMD64 [1]

У реальному режимі роботи мікропроцесори перебувають до завантаження операційної системи і можуть адресувати 1 Мбайт ОЗП, при цьому не включені механізми захисту пам’яті і рівнів доступу. У цьому режимі роботи мікропроцесори архітектури AMD64 (Intel® 64) подібні у загальних рисах до архітектури I8086.

Розглянемо у найбільш загальному вигляді архітектуру Intel 8086. Вона містить типові базові елементи: процесор, оперативну пам’ять (random access memory (RAM)), периферійні пристрої. Все це об’єднано між собою системною магістраллю. Вона включає шину адрес, шину даних і шину управління.

Кількість провідників у шині визначає її розрядність. Шина даних – 16-розрядна, шина адрес – 20-розрядна. Усі змінні, що оголошені у сегменті даних розміщуються в оперативній пам’яті і до них можливий програмний доступ. Звичайно програма розподіляється на окремі частини, що не перетинаються, які називаються сегментами. Синтаксис оголошення змінних у сегменті даних показано далі.

```
;Оголошення змінних
v1_byte     DB     1
v2_word     DW     0aaffh
v3_dword    DD     011ff11ffh
```

У цьому прикладі оголошено три змінних. Перша змінна має тип однобайтової змінної, друга змінна має тип машинного слова (розмір 2 байти), третя змінна має розмір подвійного слова (розмір 4 байти).

Для доступу до змінних, масивів, команд важливо знати способи адресації, що передбачені в архітектурі. Особливо це важливо для розроблення на Асемблері. Є різні варіанти опису способів адресації у 16-розрядній архітектурі I8086. Найбільш досконалий є варіант опису видів адресації Рудакова-Фіногенова, що подано у [4].

Відповідно до [4] усі способи адресації розділимо на такі групи:

– *безпосередня;*

– *регістрова;*

– *пряма;*

– *непряма.*

Непряма група включає: базову, індексну, базово-індексну, базово-індексну зі зміщенням. Є інші підходи до опису способів адресації, один з прикладів наведено у Дод. 1. Проте відмінність не є суттєвою.

На рис. 3.4 показано зміст дампу пам'яті, що отриманий за допомогою скриншоту під час роботи TD. Як можна побачити з рис. 3.4, у самого початку сегмента даних знаходиться двовимірний масив array2Db, що складається з однобайтових елементів і має розмір 16x16. Такий тип і розмір масиву обраний тому, що він зручно розміщується у пам'яті і гарно відображається у TD (рис. 3.4).

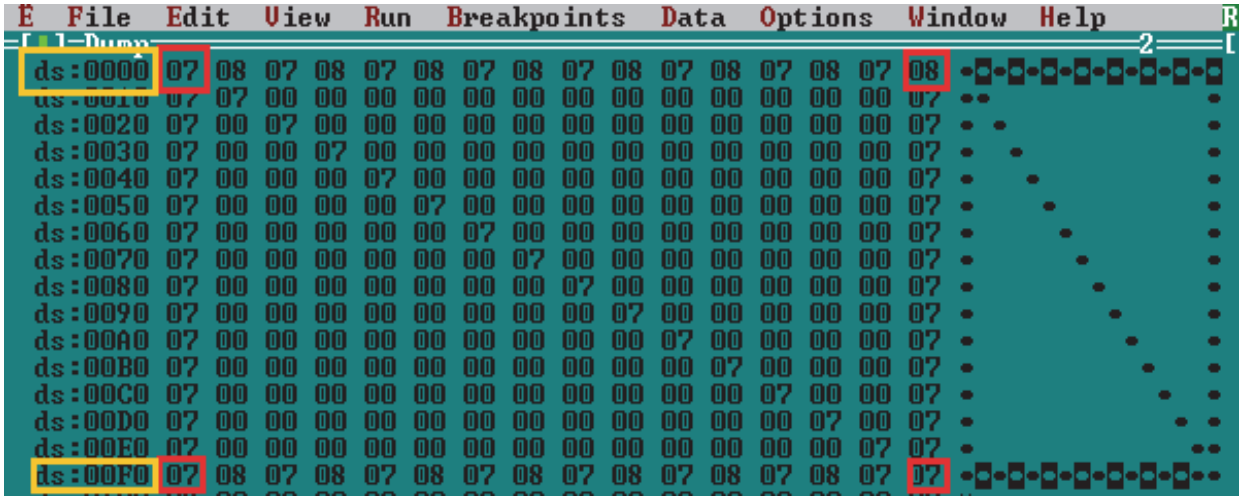



*Рис. 3.4*. Вихідний зміст пам'яті

Як можна побачити з рис. 3.4, у вікні дампу пам'яті Turbo Debugger відображає в кожному рядку зміст 16 байтів або 8 машинних слів. Ліворуч від кожного рядка показано логічну адресу першого байта у рядку. Вона складається з двох частин ds:0000 (виділено жовтим кольором). Ця логічна адреса відповідає адресі початку масиву його нульового елемента.

Вона складається з адреси початку сегмента, що знаходиться у сегменті даних, у цьому прикладі у сегментному регістрі ds.

Друга частина логічної адреси – це зміщення у сегменті. У якості пояснення визначимо логічні адреси кутових елементів масиву рис. 3.3 (виділені червоним кольором). Визначимо логічну адресу крайнього верхнього лівого байта двовимірного масиву. Для цього експерименту значення DS дорівнює 52F6. Зміщення цього елемента у сегменті дорівнює 0. Отже, його логічна адреса може бути записана DS:0000 або 52F6:0000.

У такий спосіб можна визначити логічну адресу будь-якого елемента (байта) масиву, або іншого операнда у програмі. Зрозуміло, що у реальних програмах такого зручного розміщення масиву у пам'яті не буде. Користуючись Turbo Debugger, проведемо дослідження прямої адресації. Синтаксис прямої адресації для TASM показаний у фрагменті коду нижче. У цьому прикладі для демонстрації прямої адресації до регістру AX записується число 3 і далі з використанням прямої адресації здійснюється запис до експериментального масиву.

```
; **1. Пряме звернення до пам'яті з відомою абсолютною
;адресою mov ax,03h; В ax записується константа.
; Дуже небезпечний механізм. Проте дає повну владу над кодом.
mov [DS:[00h]], ax ; Прямий запис у пам'ять за адресою DS:0000 змісту AX
mov [DS:[01h]], ax;
mov [DS:[02h]], ax;
mov [DS:[03h]], ax;
mov [DS:[04h]], ax;
mov [DS:[05h]], ax;
mov [DS:[06h]], ax;
mov [DS:[07h]], ax
```

Узагальнимо синтаксис прямої адресації до пам'яті [4].

Може бути використано безпосереднє значення, наприклад:

mov [DS:[07h]], ax.

Може бути використана назва змінної або масиву, наприклад:

mov [v1_byte], ax

Може використовуватися додатковий сегмент, або інші сегменти, наприклад сегмент коду:

mov [ES:[07h]], ax

mov [CS:[07h]], ax.

На рис. 3.5 показано результати роботи цього фрагмента коду. Для отримання цього результату виконаємо програму покроково і перевіримо ділянку дампу пам'яті з використанням TD.

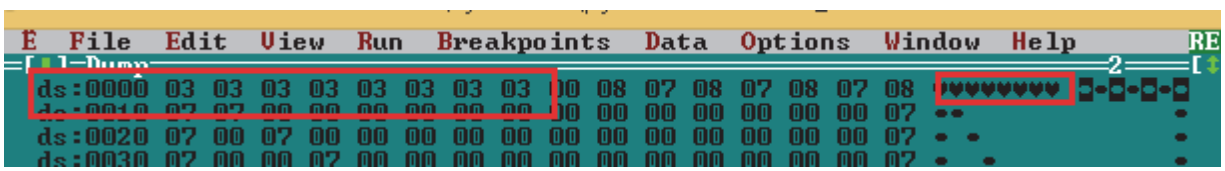



*Рис. 3.5*. Результат прямої адресації

Під час роботи програми вихідні елементи експериментального масиву було замінено значеннями регістру AX у діапазоні адрес ds:0000-ds:0007, що виділені червоним прямокутником. Таким чином експериментально підтверджено пряму адресацію.

Отже, зазначимо, що цей механізм дає можливість повної влади над пам'яттю у Real Mode і є небезпечним механізмом. Ми можемо на свій розсуд записати до будь-якої ділянки пам'яті будь-яке число, що може призвести до пошкодження даних. Цей механізм доцільно використовувати для адресації змінних.

Для зручного доступу до елементів масивів і ділянок пам'яті призначена базова й індексна адресація. Базова адресація передбачає використання для адресації регістрів BX або BP, що відповідно до синтаксису замикають у квадратні дужки. Під час використання регістру BX за замовчуванням береться логічна адреса DS:BX. Приклад синтаксису показано у фрагменті коду нижче.

Під час використання регістра BP за замовчуванням береться логічна адреса SS:BP, приклад показаний у фрагменті коду далі.

```
;**Приклад 1, базова адресація.
mov al, 02h
movbx,08h; Етап1.До BX заносимо ефективну адресу потрібної ділянки коду
mov [bx], al ;Етап 2. До пам'яті за адресою [DS]:[BX]. заносимо значення AX
incbx                ; Збільшуємо значення BX на 1
mov [bx], al ; Записуємо в інші ділянки пам'яті, все це здійснюється циклічно
incbx
mov [bx], al incbx
mov [bx], al incbx
mov [bx], al incbx
mov [bx], al incby
mov [bx], al incby
```

У молодшу частину регістру AX заносимо числове значення 2. До регістру BX заносимо зміщення (ефективну адресу), що дорівнює 08. Далі виконується ділянка коду, що показана вище. Результат роботи фрагмента коду показаний на рис. 3.6. Для нашого прикладу результат роботи виділений червоним прямокутником праворуч (рис. 3.6).

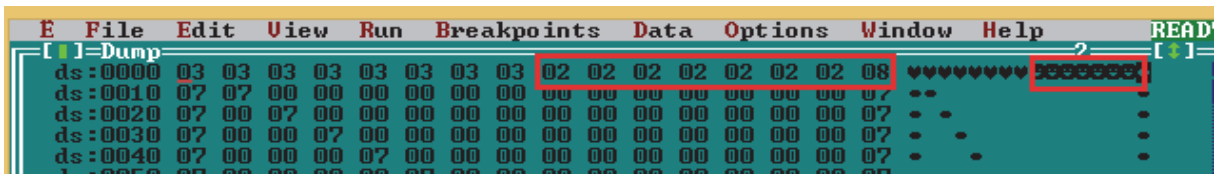



*Рис. 3.6*. Результат базової адресації

У такий спосіб експериментально підтверджена робота базової адресації. Узагальнимо синтаксис базової адресації:

mov [BX], AL
mov [BP], AL.

У цьому прикладі сегментний регістр вказувати на пряму не потрібно, за замовчуванням у разі використання BX береться логічна адреса DS:BX. У разі використання BP береться логічна адреса SS:BP.

Індексна адресація передбачає використання для адресації індексних регістрів SI або DI, що відповідно до синтаксису замикають у квадратні дужки. У разі використання SI, DI за замовчуванням береться логічна адреса DS:SI або DS:DI. Синтаксис індексної адресації показано нижче. У цьому коді регістр BX використовується як звичайний регістр загального призначення і призначений для зберігання числа, що буде записаний до ділянки пам’яті.

```
;**Приклад 1, індексна адресація.
mov di, 010h ;Визначаємо початкову адресу для запису даних
mov bx, 04h ; Записуємо число, що буде використано
mov [di],bx; M(DS*16+DI)<-BX
incdi
mov [di], bx; M(DS*16+DI+1)<-BX
inc di
mov [di], bx; M(DS*16+DI+2)<-BX
inc di
mov [di], bx; M(DS*16+DI+3)<-BX
incdi
mov [di], bx
incdi
mov [di], bx
incdi
mov [di], bx
incdi
mov [di], bx
```

До регістру DI заносимо зміщення. У нашому прикладі коду числове значення зміщення дорівнює 10h. Це буде початок другого рядка матриці. До початкового елементу другого рядка матриці записуємо число 4. Далі використовуємо нову команду inc di, що збільшує значення у регістрі DI на одиницю. Використовуємо індексну адресацію і записуємо у наступний байт пам’яті число 4. Результат багаторазового повторення цих команд показано на рис. 3.7 і виділено червоним прямокутником.

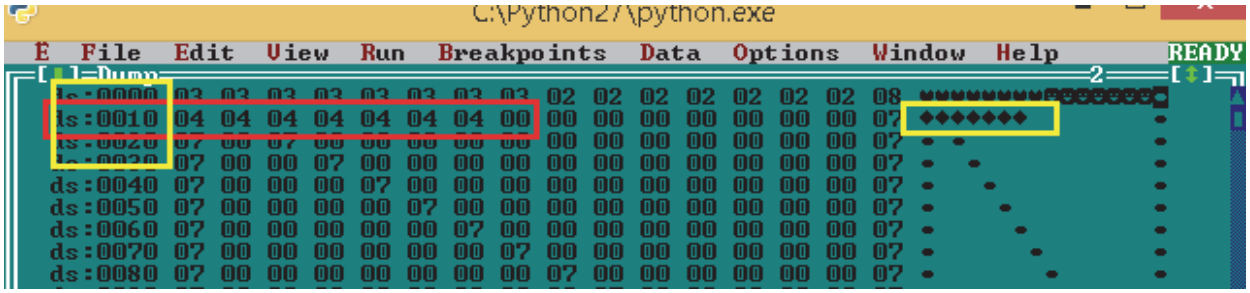



*Рис. 3.7.* Результат індексної адресації

Отже, базовий та індексний способи адресації дають можливість доступу до довільної ділянки пам'яті у сегменті даних або у сегменті стека, але такий спосіб адресації не дає зручної роботи з багатовимірними масивами.

Базово-індексна адресація дає можливість керувати пам'яттю у двох координатах дампу пам'яті. Відносна адреса операнду визначається значенням базового й індексного регістрів. Дозволено використовувати такі пари регістрів:

[BX][SI], [BX][DI], [BP][SI], [BP][DI].

Продемонструємо, як можна управляти записом до двовимірного масиву array2Db у двох координатах.

Приклад розв'язання цієї задачі з використанням базово-індексної адресації показано у фрагменті коду нижче.

```
mov al, 9                          ; Записуємо до молодшої частини AX9
mov bx,20h                  ; Готуємо базовий регістр,пересування по вертикалі
mov si,0h                          ; Готуємо індексний регістр,
пересування по горизонт. mov[array2Db+si+bx],al      ;
M(DS*16+array2Db+SI+BX)
inc si
mov
[array2Db
+si+bx],al
incsi
mov
[array2Db+
si+bx], al
add bx, 10h
mov si, 0h
mov
[array2Db
+si+bx],al
incsi
mov
[array2Db
+si+bx],al
incsi
mov [array2Db+si+bx], al
```

У якості бази обрано значення початку 3-го рядка, це значення записано до базового регістру (mov bx, 20h). До індексного регістру записано значення 0. Далі збільшується індексний регістр на одиницю, разом заповнюється третій рядок, три елементи. Далі збільшується значення базового регістру на 10h, що означає стрибок на початок наступного рядка. Аналогічно заповнюється другий рядок, результат роботи показано на рис. 3.8.

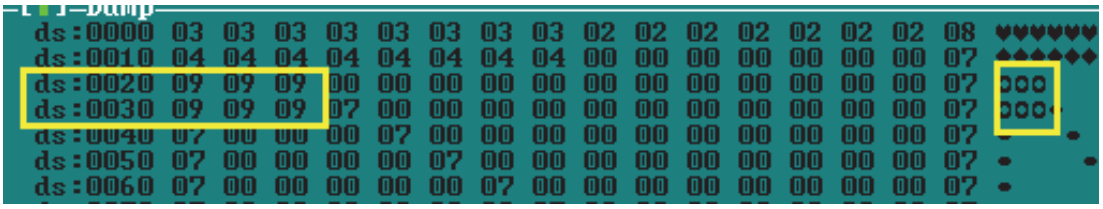



*Рис. 3.8*. Результат роботи базово-індексної адресації

Таким чином, експериментально було підтверджено можливість використання базово-індексної адресації. Крім вищеописаного передбачено базово-індексну адресацію зі зміщенням.

Як вже було згадано раніше, масиви у практичних рішеннях можуть бути розміщені в оперативній пам’яті у довільний спосіб. Наприклад, перед початком масиву може бути ділянка пам’яті. Для її урахування передбачена базово-індексна адресація зі зміщенням. Вона дає можливість додати константу-зміщення і зручно керувати пам’яттю у двох координатах. Такий спосіб адресації також є зручним для доступу до масиву, елементами якого є типи даних користувача, наприклад структури.

Для базово-індексної та базово-індексної зі зміщенням адресацій дозволяється використовувати такі пари регістрів: [BX][SI], [BX][DI], [BP][SI], [BP][DI]. Додаткове зміщення реалізується у вигляді константи. Докладного роз’яснення цей спосіб не потребує, оскільки в цілому відповідає базово-індексній адресації.

Інші види адресації – регістрова, безпосередня, стекова – особливих пояснень не потребують і описані у вихідному коді нижче. Роботу стека більш докладно буде описано у лабораторній роботі 4.

```
TITLE ЛР_3_1
;-----------------------------------------------------------------------
;ЛР  № 3-1 АК
;-----------------------------------------------------------------------
;
```

```
; Завдання:
; ВНЗ:                КПІ ім. Ігоря Сікорського
; Факультет:    ФІОТ
; Курс:    1
; Група:       __ __ __
;-------------------------------------------------------------------------------
; Автор:         __ __ __
; Дата:          __/__/__

;----------------I. ЗАГОЛОВОК ПРОГРАМИ-----------------------
IDEAL
MODEL SMALL
STACK 512
;           II. МАКРОСИ
; Складний макрос для ініціалізації
MACRO M_Init              ;Початок макросу
mov      ax, @data        ; ax <- @data
mov      ds, ax   ; ds <- ax
mov      es, ax   ; es <- ax
ENDM M_Init

;--------------------III. ПОЧАТОК СЕГМЕНТА ДАНИХ--------------
DATASEG
; Оголошення двовимірного експериментального масиву 16x16
array2Db          db 7, 8 ,7 ,8 ,7 ,8 ,7 ,8 ,7 ,8 ,7 ,8 ,7 ,8 ,7 ,8
db  7, 7 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,7
db  7, 0 ,7 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,7
db  7, 0 ,0 ,7 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,7
db  7, 0 ,0 ,0 ,7 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,7
db  7, 0 ,0 ,0 ,0 ,7 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,7
db  7, 0 ,0 ,0 ,0 ,0 ,7 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,7
db  7, 0 ,0 ,0 ,0 ,0 ,0 ,7 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,7
db  7, 0 ,0 ,0 ,0 ,0 ,0 ,0 ,7 ,0 ,0 ,0 ,0 ,0 ,0 ,7
db  7, 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,7 ,0 ,0 ,0 ,0 ,0 ,7
db  7, 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,7 ,0 ,0 ,0 ,0 ,7
db  7, 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,7 ,0 ,0 ,0 ,7
db  7, 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,7 ,0 ,0 ,7
db  7, 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,7 ,0 ,7
db  7, 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,7 ,7
db  7, 8 ,7 ,8 ,7 ,8 ,7 ,8 ,7 ,8 ,7 ,8 ,7 ,8 ,7 ,7

; Для вирівнювання у дампі
arr_def1 dw 3, 0 , 0 , 0 , 0 , 0 , 0 , 0

; Оголошення двовимірного масиву 8x8
array2Dw          dw 8, 8 ,8 ,8 ,8 ,8 ,8 ,8
dw  8, 8 ,8 ,8 ,8 ,8 ,8 ,8
dw  8, 8 ,8 ,8 ,8 ,8 ,8 ,8
dw  7, 7 ,7 ,8 ,8 ,7 ,7 ,7
dw  7, 7 ,7 ,8 ,8 ,7 ,7 ,7
dw  7, 7 ,7 ,8 ,8 ,7 ,7 ,7
dw  7, 7 ,7 ,8 ,8 ,7 ,7 ,7
dw  7, 7 ,7 ,8 ,8 ,7 ,7 ,7
; Для вирівнювання у дампі
arr_def2 dw 3, 0 , 0 , 0 , 0 , 0 , 0 , 0

; Оголошення двовимірного масиву 4x4
array2Dd          DD  34925, 34925 ,34925 ,34925
DD  34925, 30583 ,30583 ,34925
DD  34925, 30583 ,30583 ,34925
DD  34925, 34925 ,34925 ,34925
```

```
arr_def3 DW 3, 0 , 0 , 0 , 0 , 0 , 0 , 0

; Рядки повідомлень
msg_love DB "Max Natali$"
msg_asm DB "Assembler AUTS $"
msg_vb DB "Variable in byte"
; Оголошення змінних
v1_byte DB 11111111b ; однобайтова змінна
arr_def4        DB 3, 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0
msg_vw          DB      "Variable in word"
v2_word         DW 0aaffh ; змінна 1 машинне слово
arr_def5        DW 3, 0 , 0 , 0 , 0 , 0 , 0
msg_vdd         DB "Variable in dd"; 4-байтова змінна
v3_dword        DD 011ff11ffh
arr_def6        DW 3, 0 , 0 , 0
exCode DB 0

CODESEG
;----------------------VI. ПОЧАТОК СЕГМЕНТА КОДУ-------------------
Start:
M_Init
; Способи адресації за Рудаковим-Фіногеновим----------------------

; 1. Пряме звернення до пам'яті з відомою абсолютною адресою
mov ax,03h      ; В ax записується константа.
; Дуже небезпечний механізм. Проте дає повну владу над кодом.
mov [DS:[00h]], ax ; Прямий запис у пам'ять за адресою DS:0000 змісту AX
mov [DS:[01h]], ax  ;
mov [DS:[02h]], ax  ;
mov [DS:[03h]], ax  ;
mov [DS:[04h]], ax  ;
mov [DS:[05h]], ax  ;
mov [DS:[06h]], ax  ;
mov [DS:[07h]], ax
mov [v2_word], ax
; 2. Базова адресація. Призначена для роботи з масивами

mov al, 02h     ; Число, що буде записано до ділянки дампу
mov bx, 08h     ; До BX заносимо ефективну адресу потрібної ділянки коду
mov [bx], al                ; До дампу заносимо значення AX
inc bx          ; Збільшуємо значення BX на 1
mov [bx], al    ; Записуємо в інші ділянки пам'яті
inc bx
mov [bx], al    ; Записуємо в інші ділянки пам'яті,
inc bx
mov [bx], al    ; inc bx
mov [bx], al    ; inc bx
mov [bx], al    ; inc bx
mov [bx], al    ; inc bx

mov bp, 01h     ; Етап 1. До BP заносимо ефективну адресу потрібної ділянки стека
mov cx, [bp]    ; Етап 2.До CX заносимо значення з пам'яті за адресою [SS]:[BP].

; Індексна адресація. mov di, 010h
mov bx, 04h
mov [di], bx    ;M(DS*16+DI)<-BX inc di
mov [di], bx;M(DS*16+DI+1)<-BX
inc di
mov [di], bx ;M(DS*16+DI+2)<-BX
inc di
mov [di], bx ;M(DS*16+DI+3)<-BX
```

```
inc di
mov [di], bx ;M(DS*16+DI+4)<-BX
inc di
mov [di], bx ;M(DS*16+DI+5)<-BX
inc di
mov [di], bx ;M(DS*16+DI+6)<-BX
inc di
mov [di], bx ;M(DS*16+DI+7)<-BX

mov si, 01h       ; Етап 1. До SI заносимо ;значення зміщення у сегменті даних.
mov ax, [si]      ; Етап 2. До AX заносимо зміст значення з адреси пам'яті [DS]:[SI].


; Базово-індексна адресація
mov al, 9         ;Записуємо до молодшої частини AX 9
mov bx, 20h       ; Готуємо базовий регістр, пересування по вертикалі
mov si, 0h        ; Готуємо індексний регістр, пересування по горізонталі
mov [array2Db+si+bx], al;M(DS*16+ array2Db+SI+BX)
inc si
mov [array2Db+si+bx], al;M(DS*16+ array2Db+SI+BX)
inc si
mov [array2Db+si+bx], al;M(DS*16+ array2Db+SI+BX)
add bx, 10h
mov si, 0h
mov [array2Db+si+bx], al;M(DS*16+ array2Db+SI+BX)
inc si
mov [array2Db+si+bx], al;M(DS*16+ array2Db+SI+BX)
inc si
mov [array2Db+si+bx], al;M(DS*16+ array2Db+SI+BX)
;Приклад базово-індексної адресації у "чистому вигляді"
mov ax, [BP+SI]; Наприклад:
mov cx, [BP+DI]
mov dx, [BX+SI]
mov dx, [BX+DI]

; Базово-індексна й індексна адресація зі зміщенням
mov [array2Db+si+bx+3], al;M(DS*16+ array2Db+SI+BX+3)

mov si, 0         ; Ініціалізуємо регістр si – індекс, елементу масиву.
mov dl, [array2Db+si]    ; dl<- array2Db[00]
mov si, 1         ; Аналогічні дії з першим елементом масиву array2Db[1].
mov dh, [array2Db+si]   ; dh<-array2Db[10]
mov si, 2         ; Аналогічні дії з другим елементом масиву array2Db[20].
mov cl, [array2Db+si]    ; cl<- array2Db[20]


; Безпосередня адресація. Операнд входить до складу  команди
mov ax, 1         ;(ax<-1)
mov ah, 0ffh      ;( ah <-0ffh)
mov al, 0aah      ;( al <-0aah)
mov bx, 02h       ;(bx<-02h)
mov bh, 011h
mov bl, 0aah
mov dx, '7'       ;dx (dx<-"ASCII код знака 7")
mov si, 4         ;si (si<-4)
mov di, 04h       ;di (di<-04h)
mov di, 11101110b
```

```
; Стекова адресація.
; Приклад. У стеку збережені сегментні регістри
push ds; Розміщення в стеку змісту регістру ds, зміст пам'яті
; M(SSx16-SP*2)<-ds
push ss
push es
; Приклад. Відновлення значення СР
pop ds  ; Відновлення зі стека змісту регістру cs
pop ss  ; Аналогічна операція,зміст пам'яті M(SSx16+SP*2)<-ds pop es
; pop cs Не дозволено


; Регістрова адресація. Операндами є регістри. mov ds, bx
; mov ss, cx. Дозволено. Але не треба, зламає стек
Exit:


mov ah, 04Ch
mov al, [exCode] ; Вхідний параметр для функції 4ch
int 21h
END Start
```

Отже, мікропроцесори архітектури AMD64 (Intel® 64) мають два основних режими роботи [1–2]: *режим Long Mode*, що дає можливість використати всі переваги 64-розрядного режиму роботи і *режим Legacy Mode* (режим забезпечення сумісності), що призначений для підтримки 32-розрядних ОС. Залежно від режиму роботи мікропроцесорів у них доступні різні механізми адресації.

У режимі Legacy Mode є три базових режими роботи мікропроцесора, від яких залежать доступність механізмів адресації. До трьох базових режимів роботи Legacy Mode належать [1]:

– *захищений режим роботи (Protected mode*), призначений для роботи 32-розрядних ОС і у якому доступний механізм захисту пам'яті і відокремлення адресного простору різних процесів;

– *віртуальний 8086 режим роботи (Virtual-8086 mode*), у якому доступні механізми адресації архітектури I8086 для запуску цих програм у захищеному режимі;

– *реальний режим роботи (Real mode)*, призначений для 16-розрядного ПЗ, у цьому режимі процесор перебуває до завантаження основної ОС, може надавати доступ до адресації 1 Мб оперативної пам'яті. Реальний режим роботи, його механізми адресації, мають багато спільних рис з архітектурою процесорів I8086.

## *Перелік запитань для підготовки до лабораторної роботи*

1. Як програмно (TD) визначається ефективна адреса змінної (початку масиву)?

2. Опишіть пряму адресацію, синтаксис, використання. Поясніть на прикладі.

3. Опишіть регістрову адресацію, синтаксис, використання. Поясніть на прикладі коду.

4. Опишіть індексну адресацію, синтаксис, використання. Поясніть на прикладі коду.

5. Опишіть базову адресацію з BX, синтаксис, використання. Поясніть на прикладі коду.

6. Опишіть базову адресацію з BP, синтаксис, використання. Поясніть на прикладі коду.

7. Опишіть базово-індексну адресацію, синтаксис, використання. Поясніть на прикладі коду.

8. Опишіть базово-індексну адресацію зі зміщенням, синтаксис, використання. Поясніть на прикладі коду.

9. Опишіть стекову адресацію, синтаксис, використання. Поясніть на прикладі коду.

10. Назвіть перелік регістрів загального призначення архітектури Intel 8086. Опишіть призначення регістрів, особливості. Розкрийте поняття розрядності регістрів.

11. Назвіть перелік сегментних регістрів архітектури Intel 8086. Яка розрядність сегментних регістрів?

12. Опишіть оперативну пам’ять, адресу і зміст за адресою. Як здійснюється сегментація пам’яті 8086, навіщо розроблений цей механізм?

13. Що таке логічна адреса, адреса початку сегмента, зміщення у сегменті, фізична адреса операнду. Як визначається фізична адреса операнду?

14. Опишіть призначення сегмента даних, сегмента стека, сегмента коду. Синтаксис їх опису в Асемблері. Як можна визначити адресу початку сегментів з використанням TD? Поясніть на прикладі коду.

15. Опишіть синтаксис змінної, чисельного масиву, масиву символів в Асемблері. Як визначити фізичну адресу змінної, початку масиву?

# ЛАБОРАТОРНА РОБОТА 4

# ДОСЛІДЖЕННЯ РОБОТИ СТЕКА

Мета лабораторної роботи полягає у набутті впевнених знань механізму роботи стека для мікропроцесорів архітектури AMD64 (Intel® 64) у реальному режимі. Під час використання Асемблера як мови програмування застосовуються знання архітектури мікропроцесору персонального комп'ютера.

Метою лабораторної роботи є також набуття знань, умінь і навичок роботи зі стеком під час розроблення ПЗ на Асемблері у реальному режимі. Вивчення механізму доступу до довільного елементу стека з використанням базової адресації у реальному режимі.

## *Завдання на лабораторну роботу 4*

1. Створити масив arr_stack розміром 16x16, що має тип даних «слово» у сегменті даних (рис. 4.1). Заповнити масив випадковими числами. Варіанти реалізації у п. 1.1; 1.2; 1.3 за вибором студента. До одного рядка масиву додати дату народження студента, як показано на рис. 4.1. Номер рядка масиву з датою народження відповідає номеру робочої групи.

1.1. Перший варіант створення масиву (рис. 4.1) – взяти числа з Дод. 2, з таблиці випадкових чисел.

1.2. Другий варіант створення масиву (рис. 4.1) складається з двох кроків. Перший крок – написати програму на високорівневій мові і створити файл з випадковими числами. Другий крок – заповнити цими числами масив arr_stack розміром 16x16 (рис. 4.1).

1.3. Третій варіант створення масиву (рис. 4.1) – завдання підвищеної складності. Написати на Асемблері процедуру для створення псевдовипадкових чисел. Користуючись статистичними методами дослідження, визначити закон розподілу отриманих чисел. Заповнити цими числами масив arr_stack розміром 16x16 (рис. 4.1).

2. Написати процедуру, що робить кілька однакових копій цього масиву в сегмент даних. Кількість копій відповідає номеру варіанта. Для створення копій використати циклічні конструкції Асемблера або інші варіанти.

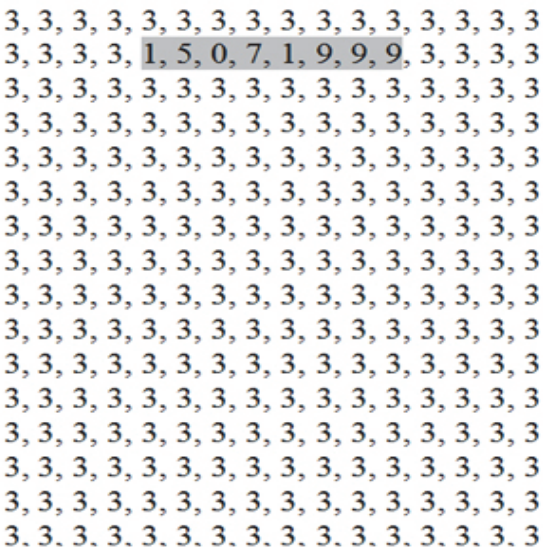



*Рис. 4.1*. Вигляд масиву

## *Програма проведення експерименту лабораторної роботи 4*

1. Створення вихідного коду. Користуючись одним з текстових редакторів, створити файл вихідного коду лабораторної роботи l4_gr1.asm. Зберегти його у робочому каталозі, що містить програмну інфраструктуру Асемблера. Записати до сегмента даних вихідного коду масив з ініціалами студентів робочої групи.

Доробити вихідний код відповідно до поданого завдання.

2. В операційній системі Windows 10 або у сучасній POSIX операційній використовувати віртуальну машину, наприклад DosBox. Запустити цю програму віртуалізації і, використовуючи консоль DosBox, перейти до робочого каталогу.

3. Далі за допомогою програми TASM.EXE провести асемблювання вихідного коду. Для асемблювання вихідного коду у командному рядку виконується така команда:

*tasm.exe /l /zil4_gr1.asm*

Докладний опис команди подано далі у теоретичних відомостях до ЛР 4.

4. Компонування або лінкування. У командному рядку виконується така команда:

*tlink.exe/v l4_gr1.obj.*

Докладний опис команди подано далі у теоретичних відомостях до ЛР 4.

5. У разі появи помилок на етапі асемблювання або лінкування внести зміни до вихідного коду і повторити етапи 3, 4.

6. Після успішного виконання п. 3, 4 провести трасування програми. Для цього необхідно виконати у командному рядку у робочому каталозі таку команду:

*td.exe l34_gr1.exe.*

7. У процесі покрокового виконання за допомогою дослідження дампу пам'яті сегмента даних і сегмента стека переконатися у правильному виконанні завдання ЛР. Це робиться за допомогою скриншотів (фотографування екранів під час роботи TD). Записати результати до звіту. Зробити висновки щодо результатів роботи програми на Асемблері.

## *Теоретичні відомості для ЛР 4*

Сучасні комп'ютери обладнуються оперативною пам'яттю обсягом від чотирьох і більше гігабайт. Процесори комп'ютера архітектури AMD64 (Intel® 64) працюють у декількох режимах роботи. У режим real mode ПК переходить до завантаження операційної системи. В цьому режимі ПК може адресувати 1 Мбайт ОЗП. Повний обсяг оперативної пам'яті комп'ютер може адресувати тільки у *режимі Long Mode*, що призначений для 64-розрядних ОС і дає можливість використати всі переваги 64-розрядного режиму роботи мікропроцесора.

У процесорах архітектури AMD64 (Intel® 64) підтримуються такі нетипізовані типи даних:

– біт (bit);

– байт (8 біт, byte);

– слово (16 біт, word);

– подвійне слово (32 біти, doubleword);

– слово «о» 4 байти (64 біти, quadword);

– подвійний слово «о» 4 байт (octword) (128 біт);

– подвійне «о» 8 байт (256 біт, double octword).

У процесорах архітектури AMD64 (Intel® 64) підтримуються апаратно беззнакові цілі числа, unsigned integer:

– 8-бітове (байтове) ціле число без знака;

– 16-бітове (слово) ціле без знака;

– 32-розрядне (doubleword) ціле число без знака;

– 64-розрядне (quadword) ціле число без знака;

– 128-розрядне (octword) ціле число без знака.

У процесорах архітектури AMD64 (Intel® 64) підтримуються апаратно знакові цілі числа, signed integer:

– 8-бітове (байтове) ціле число зі знаком;

– 16-бітове (слово) ціле число зі знаком;

– 32-розрядне (doubleword) ціле число зі знаком;

– 64-розрядне (quadword) ціле число зі знаком;

– 128-бітове (octword) ціле число зі знаком.

Також апаратно підтримуються двійкові кодовані десяткові цифри (BCD) і типи даних із плавучою комою (half-precision floating point (16 bits), single-precision floating point (32 bits), double-precision floating point (64 bits)). Узагальнений опис цілочисельних даних показано на рис. 4.1.

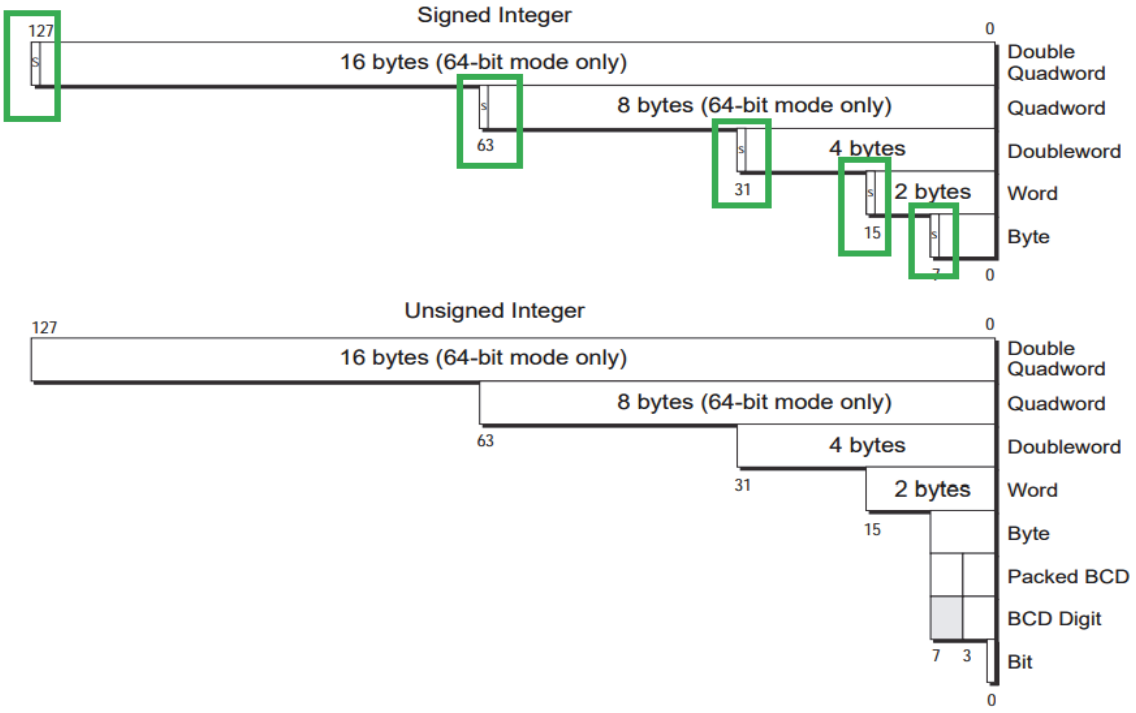



*Рис. 4.1*. Узагальнені типи даних архітектури AMD64 (Intel® 64) [1]

На рис. 4.1 зеленим кольором відокремлений біт, що призначений для індикації знака у додатковому коді.

Прикладом символьних даних у Асемблері є тип db, значення у програмі Асемблера записуються в одинарних або подвійних лапках, що проде-

монстровано у ЛР 1, кодуються за допомогою таблиці ASCII і розміщуються у пам’яті у вигляді звичайних цілих чисел зі знаком.

Якщо число у коді Асемблера у десятковій формі, то воно залишається незмінним. Для позначки числа з основою 16 у програмі Асемблера використовується буква «h», наприклад: 21h, 3A7h, 8FFFh. Якщо ж число починається з цифри-букви (A, B, C…), то перед нею треба обов’язково ставити 0, щоб Асемблер міг розрізнити, що це саме число, наприклад: 0FFF8h, 0Alh, 0C000h. Крім того, у програмі можуть використовуватися додаткові оператори для роботи із змінними й їх адресами. У синтаксисі TASM для I8086 для розміщення слова у ділянці пам’яті з визначеною адресою записується так:

mov [word ES:DI], 0E40Fh.

Аналогічний запис буде використаний для байта, подвійного слова тощо:

mov [byte ES:DI], 0E4Fh,

mov [dword ES:DI], 0EABB40h.

Ці дані можуть зберігатися у сегменті даних і сегменті стека. Стек – це програмно-апаратний елемент у архітектурі ПК. Основне його призначення – тимчасове зберігання даних. Його відрізняє найбільша швидкість під час читання і запису інформації. Саме тому він зазвичай використовується під час роботи процедур, переривань, автоматичних змінних, автоматичних масивів тощо. У високорівневих мовах це здійснюється у прихованому режимі і розробник не змушений знати особливості роботи стека.

Розробник програм на Асемблері має безпосередньо використовувати команди, що призначені для роботи зі стеком. Є дві базові команди: push [operand], pop[operand].

Як відомо з курсу лекцій, програма на Асемблері у більшості випадків складається з трьох сегментів, а саме: сегмента коду, сегмента даних, сегмента стека. Можуть використовуватися додаткові сегменти. Кожному сегменту відповідають сегментні регістри, наприклад для зберігання початку сегмента стека використовується сегмент SS. Для архітектури I8086 на початку програми директивою визначається розмір сегмента стека, наприклад STACK 512, тобто розмір 512 байт або 200h. Це означає, що у сегменті стека резервується 200h байт пам’яті.

На вершину стека вказує покажчик стека регістр SP. У порожньому стеку покажчик вказує на логічну адресу SS:0200h для наведеного прикладу. Під час виконання команди push [operand], операнд додається до стека і покажчик вказує на нову вершину стека з цим операндом. У фрагменті коду, що подано далі (Дод. 1), показано, як змінюється адреса вершини стека.

```
push   ax              ; M(SSx16+SPпоч-1)<-ah
;      M(SSx16+SPпоч-2)<-al
;      SP<-SPпоч-2
push   bx              ;      M(SSx16+SPпоч-1)<-bh
;      M(SSx16+SPпоч-2)<-bl
;      SP<-SPпоч-2

pop    ax              ;      al<-M(SSx16+SPпоч)
;      ah<-M(SSx16+SPпоч+1)
;      SP<-SPпоч+2
pop    bx              ;      bl<-M(SSx16+SPпоч)
;      bh<-M(SSx16+SPпоч+1)
;      SP<-SPпоч+2
```

Після виконання цієї команди регістр SP містить нову адресу (рис. 4.2 правий нижній кут виділено жовтим прямокутником). Вона утворюється за допомогою віднімання від попереднього значення числа 2. У сегментному регістрі SS міститься адреса початку сегмента стека. Під час зростання стека, що відбувається у разі виконання команди push AX, значення, покажчика стека SP ЗМЕНШУЄТЬСЯ на 2, і у відповідну ділянку стека записується значення операнду, що завжди має розмір одного слова (в нашому прикладі це зміст регістра AX).

Стек зростає, хоча адреса у SP покажчику стека зменшується на 2 байта (рис. 4.2). Приклад команди push показано на рис. 4.2 і виділено коричневим прямокутником, значення покажчика стека зменшується на 2 і дорівнює 01FEh, виділено жовтим кольором. Значення SP (виділено блакитним) містить 01FEh, що на 2 менше ніж 0200h, а в сегмент стека розміщується значення AX (виділено червоним).

Такий спосіб організації стека характерний саме для процесорів архітектури i8086 і аналогічних до них архітектур. Для будь-якого виду МПС треба запам’ятати, що під час виконання команди push[operand], вважається, що стек зростає. Крайній елемент на вершині стека доступний команді pop [operand].

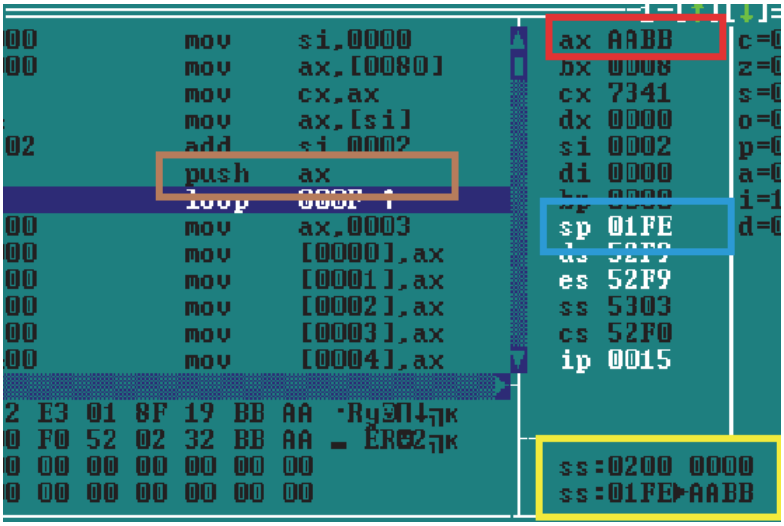



*Рис. 4.2*. Виконання команди push AX

Операнд може бути вийнятий зі стека командою pop [operand]. У цьому випадку до операнду записується зміст вершини стека. Покажчик стека зсувається з вершини нижче, при цьому для архітектури Intel 8086, після виконання команди pop [operand], адреса вершини стека (зміст SP) збільшиться на 2. Стек зменшується, а адреса вершини збільшується, що іноді призводить до плутанини.

Таким чином, *для будь-якого виду МПС треба запам'ятати, що під час виконання команди pop [operand] вважається, що стек зменшується, зміст вершини стека розміщується в [operand], а вершина стека пересувається на наступний нижчий елемент, що знаходиться у стеці.*

*Під час виконання команди push [operand] вважається, що стек зростає, зміст операнду розміщується у вершину стека.*

Для виконання ЛР 4 потрібно знати, як відбувається робота циклів в Асемблері. Розглянемо перший приклад циклу у канонічному варіанті користуючись фрагментом коду, що подано нижче. У результаті роботи цього коду масив з логічною адресою нульового елементу ds:[si] заповнюється числами 3.

```
;
;Типова організація циклу, індексна адреса
mov al, 03h
mov cx, 100h ;Заповнення усього масиву
ptr_1:
mov [ds:[si]], al inc si
loop ptr_1
```

*Як це працює*. У прикладі використана канонічна організація циклу з використанням команди loop ptr_1. Ця команда буде переводити управління на мітку ptr_1 поки значення регістру CX не дорівнюватиме 0. У коді використано індексну адресацію. До регістру CX записано значення 100h, тобто число повторень у циклі. Інших маніпуляцій з CX не потрібно, він буде зменшуватися на 1 у кожній ітерації циклу автоматично. В цьому полягає особливість регістру CX, яка властива тільки йому. Використаємо цю властивість і продемонструємо роботу команди push на прикладі коду, що показано нижче.

```
; Робота стека, команда push
    lea si, [array2Dw]
    mov ax, [array2Dwlen]
    mov cx, ax
stack1:
    mov ax, [si]
    add si, 2
    push ax
    loop stack1
```

Розглянемо більш докладно, як працює фрагмент програми для демонстрації роботи команди push. Цей фрагмент забезпечує розміщення вихідного масиву array2Dw, що знаходиться в сегменті даних до сегмента стека повністю. Перші три команди,

```
lea si, [array2Dw]
mov ax, [array2Dwlen] mov cx, ax,
```

виконують такі функції: до SI записується зміщення [array2Dw], розміщення AX довжини масиву [array2Dw], копіювання AX до CX, що є підготовкою до організації типового циклу loop. Наступні чотири команди:

```
stack1:
mov ax, [si]
add si,2
push ax
loop stack1
```

виконують такі функції: до AX розміщується значення 0 елементу масиву, перехід на другий елемент масиву за рахунок збільшення SI на 2, зменшення CX на 1 (у фоні), нова ітерація циклу і нарешті запис до стека 0 елементу. Ці дії виконуються циклічно, кількість виконання дорівнює довжині масиву.

У такий спосіб усі елементи вихідного масиву розміщуються у сегменті стека. Результат роботи програми показано на рис. 4.3.

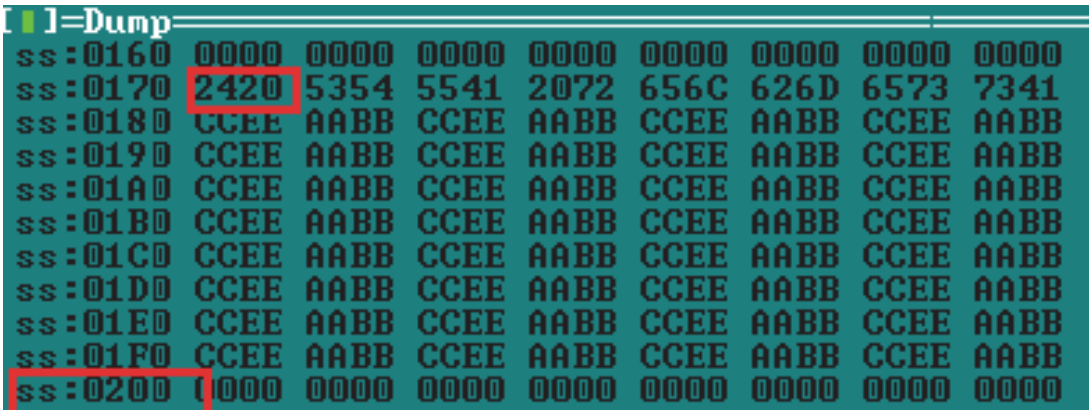



*Рис. 4.3*. Результат роботи програми

Червоним виділено початок стека і вершину стека після виконання фрагменту коду. У стеку знаходиться масив повністю. Крім масиву до стека розміщено додаткові значення і вершина стека представлена числом 2420.

Проведемо дослідження зворотної операції pop. Для дослідження показано фрагмент коду

```
    ; Дослідження команди
    lea si, [arrayRes]
    mov cx, 64

stack2:
    pop ax
    mov [si], ax
    add si, 2
    loopstack2
```

*Пояснимо фрагмент коду*. Перші дві команди призначені для підготовки елементів масиву. До SI розміщується зміщення масиву, до якого буде здійснюватися запис інформації з сегмента стека командою pop. До CX розміщується кількість елементів у масиві.

Далі у тілі циклу виконується команда pop AX. Після цієї команди у регістрі AX розміщується значення вершини стека, а покажчик стека заглиблюється на один елемент (одне слово). Далі зміст AX записується до масиву arrayRes. І цикл повторюється під час виконання команди pop, адреса у SP покажчику стека збільшується на 2. У нашій архітектурі це ознака зменшення стека. Результат роботи програми показано на рис. 4.4.

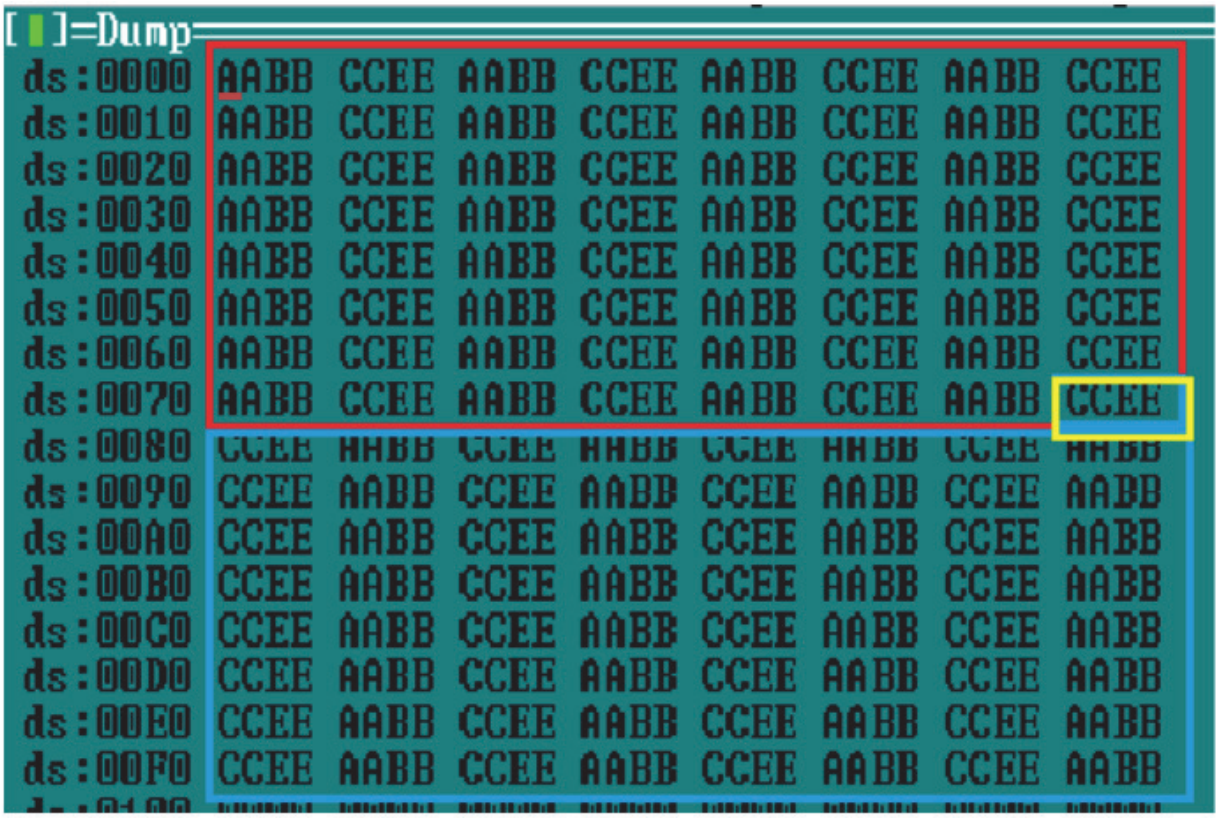



*Рис. 4.4*. Результат роботи програми, зміст сегмента даних

У червоному прямокутнику показаний масив, що був розміщений у стеку і був записаний до сегмента даних.

Внаслідок роботи програми до нового масиву було внесено значення зі стека. Першим елементом став елемент, що розміщений у вершині. Внаслідок цього елементи вихідного і нового масивів помінялися місцями.

Архітектура I8086 дозволяє довільний доступ до стека з використанням базової адресації. Для цього наведемо фрагмент коду, у якому до стека через базову адресацію записано число 0A0Ah.

```
; Дослідження базової адресації для доступу до стека
mov bp, 0200h; Вершина стека
mov [bp], 0A0Ah
```

*Пояснимо фрагмент коду*. У першому рядку коду запишемо число 200h до базового регістру BP. Далі, користуючись правилами базової адресації, розмістимо до стека значення 0A0Ah, виконується команда mov [bp], 0A0Ah. У цьому прикладі не вимагається явно вказувати сегментний регістр стека. Нижче наведено повністю вихідний код, що демонструє роботу стека.

```
TITLE ЛР_4_1
;-------------------------------------------------------------------------
;ЛР  № 4-1 АК
;-------------------------------------------------------------------------
;
; Завдання:
; ВНЗ:                КПІ ім. Ігоря Сікорського
; Факультет:    ФІОТ
; Курс:    1
; Група:       __ __ __
;-------------------------------------------------------------------------
; Автор:        __ __ __
; Дата:         __/__/__
;--------------------------------
;---------------I. ЗАГОЛОВОК ПРОГРАМИ----------------
IDEAL
MODEL SMALL
STACK 512

MACRO M_Exit
; На виході:AL = код завершення програми
; На вході:AH = ознака переривання DOS виходу 04Ch
mov   ah, 04Ch
int 21h
ENDM M_Exit


MACRO M_Init
mov ax, @data; ax <- @data
mov  ds, ax      ; ds <- ax
mov  es, ax      ; es <- ax
ENDM M_Init   ;

DATASEG

array2Dw dw 0AABBh, 0CCEEh ,0AABBh ,0CCEEh, 0AABBh, 0CCEEh ,0AABBh ,0CCEEh
dw 0AABBh, 0CCEEh ,0AABBh ,0CCEEh, 0AABBh, 0CCEEh ,0AABBh ,0CCEEh
dw 0AABBh, 0CCEEh ,0AABBh ,0CCEEh, 0AABBh, 0CCEEh ,0AABBh ,0CCEEh
dw 0AABBh, 0CCEEh ,0AABBh ,0CCEEh, 0AABBh, 0CCEEh ,0AABBh ,0CCEEh
dw 0AABBh, 0CCEEh ,0AABBh ,0CCEEh, 0AABBh, 0CCEEh ,0AABBh ,0CCEEh
dw 0AABBh, 0CCEEh ,0AABBh ,0CCEEh, 0AABBh, 0CCEEh ,0AABBh ,0CCEEh
dw 0AABBh, 0CCEEh ,0AABBh ,0CCEEh, 0AABBh, 0CCEEh ,0AABBh ,0CCEEh
dw 0AABBh, 0CCEEh ,0AABBh ,0CCEEh, 0AABBh, 0CCEEh ,0AABBh ,0CCEEh
array2Dwlen = $-array2Dw

arrayRes dw '*' dup (64)
arrayReslen = $ - arrayRes
msg_asm DB "Assembler  $"
exCode db 0
CODESEG
;       I. ПОЧАТОК СЕГМЕНТА КОДУ
Start:
M_Init
```

```
; Дослідження базової адресації для доступу до стека
mov bp, 0200h; Вершина стека
mov [word bp], 0A0Ah
; Дослідження команди push
lea si, [array2Dw]
mov ax, [array2Dwlen]
        mov cx, ax
stack1:
mov ax, [si]
        add si, 2
push ax
loop stack1

; Дослідження команди pop
lea si, [arrayRes]
mov cx, 64

stack2:
pop ax
mov [si], ax
add si, 2
loop stack2
;           ===== управління пам'яттю==== nop
nop
;*виділення пам'яті*
mov ah, 048h ; Ознака переривання
mov bx, 02h ; Розмір нової ділянки у параграфі
;*Parameters for output*
; CF = 0 if ok
; AX Адреса нового сегмента пам'яті
; CF = 1 Ознака помилки виконання переривання
; AX = 7 Ознака помилки
; AX = 8 Ознака малого обсягу пам'яті
; BX розмір нової ділянки
int 21h
; Перенесення масиву до нової ділянки пам'яті
mov es, ax      ; Ініціалізація сегментного регістру новою адресою
mov cx, 8       ; Визначення розміру нової ділянки масиву множенням
mov ax, 8
mul cx
mov cx, ax      ; Підготовка циклу до виконання, кількість ітерацій
xor di, di
lea si, [array2Dw] ; Ефективна адреса масиву до індексного рег.

mem1:
mov dx, [si]    ; Нульовий елемент у регістр. Індексна адресація
mov [es:di], dx          ;Запис до нової ділянки, індексна адресація
add si, 2       ; Перехід на новий елемент масиву
add di, 2
loop mem1
```

```
nop
nop
;**Зміна розміру ділянки. Вхідні параметри
mov ah, 04Ah  ;Ознака команди
mov bx, 01h    ;Новий розмір
; ES адреса сегмента, що буде змінюватися
;***Вихідні параметри***
; CF = 1 if not ok
; AX = 7 if memory blocks is destroyed
; AX = 9 Некоректна адреса
; AX = 8 Невистачає памяті
; BX розмір нової пам'яті
int 21h

nop
nop
;**Звільнення пам'яті. Вхідні параметри***
mov ah, 049h ; command mark
; ES адреса блоку, що звільняється
;***Вихідні параметри***
; CF = 0 ok
; CF = 1, AX = 7 if memory blocks is destroyed
; CF = 1, AX = 9 Некоректна адреса

int 21h
Exit:
mov ah,4ch
mov al,[exCode]
int 21h
end Start
```

Для управління пам'яттю і виділення додаткової ділянки, що призначена для роботи з даними в операційній системі DOS в Асемблері є переривання DOS. Вони є аналогом стандартних функцій мови програмування С, для виділення пам'яті. Наприклад функція С malloc – для виділення, для зміни обсягу пам'яті функція realloc, для звільнення ділянки пам'яті free.

Розглянемо перше переривання для виділення пам'яті. Для цього використаємо фрагмент вихідного коду, що подано далі.

```
mov ah, 048h ; Ознака переривання
mov bx, 02h ; Розмір нової ділянки у параграфі
;***Parameters for output***
; CF = 0 if ok
; AX Адреса нового сегмента пам'яті
; CF = 1 Ознака помилки виконання переривання
; AX = 7 Ознака помилки
; AX = 8 Ознака малого обсягу пам'яті
; BX розмір нової ділянки
int 21h
```

У цьому фрагменті показано приклад виклику переривання DOS і виділення ділянки пам’яті розміром у 2 параграфи. Результат роботи цієї ділянки програми можна пояснити на рис. 4.5.

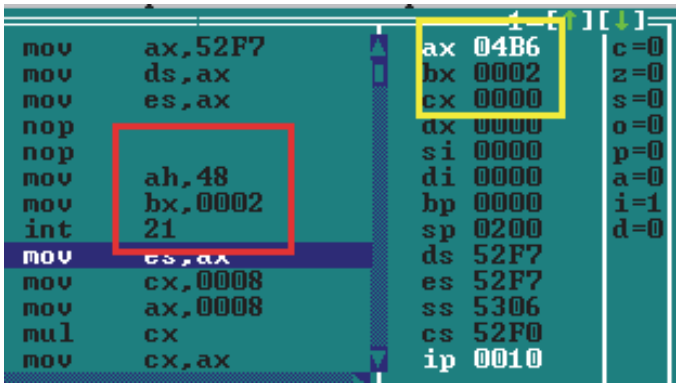



*Рис. 4.5*. Результат виділення пам’яті

Червоним кольором виділені команди і переривання для надання ділянки пам’яті. Жовтим кольором виділені регістри, що є вихідними для цього переривання. Перенесення масиву до нової ділянки пам’яті показано далі у вихідному коді.

```
;
moves,ax          ; Ініціалізація сегментного регістру новою адресою
movcx,8           ; Визначення розміру нової ділянки масиву множенням
mov ax, 8 mul cx
movcx,ax          ; Підготовка циклу до виконання, кількість ітерацій
xor di, di
lea si, [array2Dw] ; Ефективна адреса масиву до індексного рег.

mem1:
movdx, [si]       ; Нульовий елемент у регістр. Індексна адресація
mov[es:di],dx     ; Запис до нової ділянки, індексна адресація
addsi,2           ; Перехід на новий елемент масиву
add di, 2
loop mem1
```

Інші функції управління пам’яттю особливих пояснень не потребують, оскільки докладно описані у вихідному коді і мають відповідні коментарі.

Отже, для мікропроцесорів архітектури AMD64 (Intel® 64) у реальному режимі робота стека подібна архітектурі I8086. У інших режимах роботи архітектури AMD64 (Intel® 64), починаючи від захищеного режиму, кожний процес має власний стек з відокремленим адресним простором.

У реальному режимі роботи для доступу до довільного елементу стека використовується базова адресація з регістром BP.

## *Перелік запитань для підготовки до лабораторної роботи*

1. Опишіть загальну схему роботи «переривання» в архітектурі комп’ютера. Які види переривань ви знаєте?

2. Розкрийте поняття програмного переривання, переривання DOS і BIOS, як використовується це переривання. Поясніть на прикладі коду користуючись довідковою літературою.

3. Розкрийте поняття апаратного переривання. Що таке масковані й немасковані переривання. Опишіть процес переривання мікропроцесора 8086 користуючись структурною схемою.

4. Яке призначення контролера переривань, як здійснюється програмне управління контролером?

5. Що таке таблиця векторів переривань, у якій ділянці пам’яті вона знаходиться?

# ЛАБОРАТОРНА РОБОТА 5
# УПРАВЛІННЯ ПРОЦЕСОМ ВИКОНАННЯ ПРОГРАМИ НА АСЕМБЛЕРІ В АРХІТЕКТУРІ AMD64 (Intel® 64) У REAL MODE

Мета лабораторної роботи полягає у вдосконаленні знань, умінь та навичок з технології розроблення ПЗ на Асемблері для мікропроцесора архітектури AMD64 (Intel® 64) у реальному режимі.

Метою також є підготовка студента для розроблення складного програмного забезпечення з використанням Асемблера, вивчення додаткових команд Асемблера, що дають можливість керувати програмою, та додаткових переривань DOS і BIOS для відображення інформації на консолі.

## *Завдання на лабораторну роботу 5*

1. Користуючись результатами попередніх лабораторних робіт, заповнити середину масиву 16x16 ділянкою з датами народження студентів робочої підгрупи (див. рис. 5.1).

```
3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3
3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3
3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3
3, 3, 3, 0, 0, 0, 0, 0, 0, 0, 0, 3, 3, 3, 3, 3
3, 3, 3, 2, 7, 0, 7, 1, 9, 7, 7, 3, 3, 3, 3, 3
3, 3, 3, 0, 5, 7, 7, 1, 9, 7, 7, 3, 3, 3, 3, 3
3, 3, 3, 0, 8, 2, 7, 1, 9, 7, 7, 3, 3, 3, 3, 3
3, 3, 3, 0, 9, 9, 3, 1, 9, 7, 7, 3, 3, 3, 3, 3
3, 3, 3, 0, 0, 7, 5, 1, 9, 7, 7, 3, 3, 3, 3, 3
3, 3, 3, 0, 0, 7, 5, 1, 9, 7, 7, 3, 3, 3, 3, 3
3, 3, 3, 0, 0, 7, 5, 1, 9, 7, 7, 3, 3, 3, 3, 3
3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3
3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3
3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3
3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3
3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3
```

*Рис. 5.1*. Вигляд масиву

Ділянка з датами народження має розмір 8x8. У ділянку розмістити числа дня, місяця, року народження студента. Наприклад даті народження 27.07.1977 відповідає рядок на 8 елементів:

2, 7, 0, 7, 1, 9, 7, 7, де кожна цифра відповідає даті, місяцю і року народження.

2. Координати початку ділянки 8x8 (верхнього лівого кута) мають відповідати варіанту.

3. Створити процедуру Асемблера, що робить сортування масиву для парного варіанта за зростанням, для непарного варіанта за зменшенням.

За бажанням студента написати фрагмент коду, що здійснює виведення масиву на консоль з використанням прямого доступу до пам'яті або функцій DOS або BIOS.

### *Програма проведення експерименту лабораторної роботи 5*

1. Створення вихідного коду. Користуючись одним з текстових редакторів, створити файл вихідного коду лабораторної роботи l5_gr1.asm. Зберегти його у робочому каталозі, що містить програмну інфраструктуру Асемблера.

Обирається потрібна координата ділянки масиву 8x8 і, використовуючи потрібний вид адресації, пишеться код, що дає можливість заповнити потрібну ділянку коду цифрами, що вимагаються.

Отже вихідний код дороблюється відповідно до поданого завдання.

2. В операційній системі Windows 10 або у сучасній POSIX операційній використовувати віртуальну машину, наприклад DosBox. Запустити цю програму віртуалізації і, використовуючи консоль DosBox, перейти до робочого каталогу.

3. Далі за допомогою програми TASM.EXE провести асемблювання вихідного коду. Для асемблювання вихідного коду у командному рядку виконується така команда:

*tasm.exe /l /zil5_gr1.asm*

Докладний опис команди подано далі у теоретичних відомостях до ЛР 5.

4. Компонування або лінкування. У командному рядку виконується така команда:

*tlink.exe/v l5_gr1.obj.*

Докладний опис команди подано далі у теоретичних відомостях до ЛР 5.

5. У разі появи помилок на етапі асемблювання або лінкування внести зміни до вихідного коду і повторити етапи 3, 4.

6. Після успішного виконання п. 3, 4 провести трасування програми. Для цього необхідно виконати у командному рядку у робочому каталозі таку команду:

*td.exe l5_gr1.exe.*

7. У процесі покрокового виконання за допомогою дослідження дампу пам'яті сегмента даних і сегмента стека переконатися у правильному виконанні завдання ЛР. Це робиться за допомогою скриншотів (фотографування екранів під час роботи TD).

8. Якщо обраний варіант виведення інформації на екран, потрібно запустити програму і зробити скриншоти екрана.

Записати результати до звіту. Зробити висновки щодо результатів роботи програми на Асемблері.

## *Теоретичні відомості для ЛР 5*

У синтаксисі TASM для запису слова у пам'ять використовується код, що показано нижче:

```
mov [word ES:DI], 0040Fh.
```

Аналогічний запис буде використано для байта подвійного слова тощо:

```
mov [byte ES:DI], 0E4Fh,
mov [dword ES:DI], 0EABB40h.
```

У МПС можуть оброблятися беззнакові числа і числа зі знаком. Для роботи з такими числами старший біт числа зарезервовано для зберігання знака. Отже, для зберігання чисел зі знаком використовують такі самі регістри, тільки старший біт регістра (слова) резервується для зберігання знака. У цьому випадку наявність у старшому біті одиниці означає, що це від'ємне число, в іншому випадку вважається, що це позитивне число.

Для практичного виконання у процесорах використовується додатковий код, який існує для від'ємних чисел. Під час використання додаткового коду операцію віднімання замінюють операцією додавання. Для отримання додаткового коду певного від'ємного числа його значення потрібно інвертувати і додати одиницю.

Наприклад FFFBh, можна розглядати як позитивне, так і негативне, залежно від контексту використання. Таким чином, знак числа є характеристикою не самого числа, а контекст використання і команд Асемблера. Процесор може виконувати операції не тільки над словами, але й над байтами. Число у байті можна розглядати як беззнакове, що має діапазон від 0 до 255, або як число зі знаком. Тоді діапазон позитивних значень зменшується у два рази (від 0 до 127 позитивних чисел, –1 до –128 від'ємних чисел).

Нечутливими до знака є команди INC, DEC, TEST, MUL, DIV, JA, JB та ін. Ці команди будь-яке число «бачать» як позитивне, незалежно від стану старшого біту у слові або регістрі. Є група команд Асемблера, що спеціально призначена для оброблення чисел зі знаком (IMUL, IDIV, JG, JL і т. д.). Для цих команд процесор обов'язково «дивиться» у старший біт регістра або слова і визначає знак числа.

Для прикладу розглянемо команди множення. Для множення беззнакових чисел використовується MUL (multiplication, множення), для чисел зі знаком команда – IMUL (integer multiplication). Обидві команди можуть працювати як зі словами, так і з байтами. Вони виконують множення числа, що перебуває в регістрах AX (у випадку множення на слово) або AL (у випадку множення на байт). Другим операндом може бути регістр або комірка пам'яті. Не допускається множення на безпосереднє значення, а також на вміст сегментного регістра. Розмір результату в них завжди у два рази більше розміру співмножників. Для 1-байтових операцій отриманий добуток записується до AX. Для 2-байтових операцій результат множення, що має розмір 32 біта, записується до DX і AX (у DX – старша половина, в AX – молодша).

Більшість програм вимагають зміни лінійної послідовності виконання операторів і переходу на інші частини програмного коду. Також вимагається організація циклічних операцій і переходу в різні місця коду. Для цього використовуються команди загального призначення (general-purpose instructions), що можна розділити на кілька груп:

– команди переміщення (пересилання, передачі) даних;

– команди арифметики (додавання, вирахування, множення й ділення) з цілими числами;

– команди логічних операцій;

– команди передачі керування (умовних і безумовних переходів, виклик процедур);

– команди строкових операцій (іноді трапляється назва «строкові», або «ланцюгові» команди).

Розглянемо докладніше групу команд для передачі управління. Ця важлива група команд розподіляється на підгрупи. Перша розрізняє числа зі знаком, друга – числа без знака, третя – це команди умовних переходів. Ці команди дозволяють утворювати циклічні операції й операції переходів. Для цього команди умовних переходів зазвичай використовують після команд порівняння (CMP), інкремента (INC), декремента (DEC), додавання (ADD), вирахування (SUB), перевірки (TEST) та інших.

Наведемо перелік команд умовних переходів.

*Знакові команди:*

jg (jumpif greater – перехід, якщо більше);

jge (jump if greater or equal o – перехід, якщо більше або дорівнює);

jl (jumpifless – перехід, якщо менше);

jng (jump if not greater - перехід, якщо не більше);

jnge (jump if not greater or equal – перехід, якщо не більше й не дорівнює);

jnl (jump if not less – перехід, якщо не менше);

jnle (jump if not less or equal – перехід, якщо не менше й не дорівнює).

*Беззнакові команди:*

ja (jump if above – перехід, якщо вище);

jae (jump if above or equal – перехід, якщо вище або дорівнює);

jb (jump if below – перехід, якщо нижче);

jbe (jump if below or equal – перехід, якщо нижче або дорівнює);

jna (jump if not above – перехід, якщо не вище);

jnae (jump if not above ore qual – перехід, якщо не вище й не дорівнює);

jnb (jump if not below – перехід, якщо не нижче).

Приклади команд, *не чутливих до знака числа*:

je (jump if equal – перехід, якщо дорівнює);

jne (jump if note qual – перехід, якщо не дорівнює);

jc (jump if carry – перехід, якщо прапорець CF установлений);

jcxz (jump if CX=0 перехід, якщо CX = 0).

Різниця між знаковими й беззнаковими командами умовних переходів полягає в тому, що знакові команди розглядають старший біт операнду як знаковий біт. Беззнакові команди розглядають абсолютне значення операнду.

Команда безумовного переходу JMP передає управління до визначеної у програмі мітки. Саме ця команда дає можливість керувати процесом виконання програми. Команда не впливає на прапорці процесора. Команди переходів мають декілька різновидів: перехід прямий і короткий (у межах –128...+127 байт); перехід прямий і ближчий (у межах поточного сегмента команд); перехід прямий і дальній (в інший сегмент команд); перехід непрямий і ближчий; перехід непрямий дальній. У деяких випадках Асемблер може визначити вид переходу за контекстом, у деяких випадках для цього використовуються спеціальні атрибути, наприклад:

– SHORT – перехід прямий короткий (у межах –128...+ 127 байт);

– NEAR PTR – перехід прямий ближчий (у межах поточного сегмента команд);

– FAR PTR – перехід прямий дальній (в інший сегмент команд);

Команда LEA завантажує до першого операнду ефективну адресу другого операнду, що визначається у вигляді ділянки пам’яті, змінної, початку масиву, наприклад

LEA REG, MEM.

Команда не впливає на прапорці процесора. Розглянемо приклад використання команди фрагментом коду:

lea dx, [no_ziro].

Після виконання цієї команди до регістру dx буде записано зміщення масиву (або змінної) no_ziro.

Розглянемо типові конструкції для розгалуження й організації циклів. Організацію розгалужень у програмах на Асемблері найкраще пояснити на прикладі. У наступному фрагменті програмного коду виконується перехід на мітку next у разі рівності нулю вмісту регістра CX. Рівність нулю вмісту регістра CX визначається за допомогою команди cmp, що впливає на прапорці AF, CF, OF, PF, SF і ZF. Отже, наведемо приклад:

```
; Організація конструкції розгалуження (cmp)
        mov cx, 0
        cmp cx, 0
        jznext_z1
        ; оброблення ситуації, коли CX не дорівнює O
        mov ah, 09h
        lea dx,[no_ziro]
        int21h
        jmp next_2
next_z1:
        ; оброблення ситуації, коли CX дорівнює O
        mov ah, 09h
        lea dx, [is_ziro]
        int 21h
        jmp exit_3
next_2:
        mov ah, 09h
        lea dx, [out_of]
        int 21h
        exit_3:
```

*Як це працює*. Для цього прикладу, якщо CX містить нульове значення, команда CMP установлює прапорець нуля ZF в одиницю. Команда JZ перевіряє прапорець ZF і, якщо він дорівнює 1, передає управління на мітку next_z1. Після виконання коду, відбувається вихід з конструкції. В протилежному випадку виконуються код на jmp next_2, відбувається вихід назовні конструкції. Фактично цей фрагмент програмного коду реалізує оператор if у C подібних мовах програмування, з умовою CX = 0.

Аналогічно можна реалізувати оператор if з використанням команди TEST. Ця команда виконує операцію логічного «І» над двома операндами. Не змінює жодного з операндів й залежно від результату встановлює прапорці SF, ZF і PF, при цьому прапорці OF і CF скидаються, а прапорець AF має невизначене значення. Наведемо приклад з вихідного коду.

```
        mov ax, 0
; Code organization with TEST. Analog IFtest ax, 10000000b
        jne next_z2
        ; Code for bit of AX IS 1mov ah, 09h
        lea dx, [no_ziro]
        int 21h
        jmp next_1
next_z2:
; Code for bit of AX IS 0 mov ah, 09h
lea dx,[is_ziro]
int21h
```

```
next_1:
mov ah, 09h
lea dx,[out_of]
int21h
```

Можливі інші модифікації вихідного коду Асемблера для алгоритму розгалуження, але їх суть залишається тією самою: на першому етапі відбувається виконання команди TEST або CMP для перевірки умови, далі використовуються команди умовного переходу і безумовного переходу на мітки.

Умовні переходи використовуються також під час програмування циклічних алгоритмів на Асемблері. Цикл може закінчитися в одному із двох випадків: у разі виконання всіх ітерацій; коли виявлена умова, відповідно до якої повинен відбутися вихід із циклу.

Наступна важлива група команд – це строкові команди. Вони значно спрощують і пришвидшують процес запису великих обсягів інформації з однієї ділянки пам'яті до іншої. Це здійснюється без використання циклів. У складі команд процесора Intel 8086 ця група призначена для операцій з рядками символів або чисел, тобто з масивами даних. Таких команд усього 5:

1) movs – пересилання рядка;

2) cmps – порівняння рядків;

3) seas – пошук у рядку заданого елементу (сканування рядка);

4) lods, (b,w,d) – завантаження з рядка регістрів AX або AL;

5) stos, (b,w,d) – запис елементу рядка з регістрів AX або AL.

Команди мають загальні риси. Вони виконуються процесором у припущенні, що логічна адреса рядка-джерела визначається групою регістрів DS:SI, а логічна адреса рядка-приймача групою регістрів ES:DI. Під час однократного виконання вони обробляють тільки один елемент масиву. Для оброблення рядка повинні мати префікс повторення. У процесі оброблення елементи регістрів SI й DI автоматично зсуваються вперед (якщо прапорець встановлений DF = 0). Можуть зсуватися назад, якщо прапорець DF = l має значення одиниці.

Кожна команда має модифікації для роботи з байтами або словами (наприклад, movsb imovsw). Розглянемо особливості використання строкових команд на простих прикладах з вихідного коду нижче.

```
; Запис великих масивів без організації циклів
; Робота з рядками джерело (sourse) у DS:SI, приймач у ES:DI

cld                     ; Обнулення прапорців напрямку, прямий
leasi,[array2Db]        ; Завантаження відн. адреси джерела
lea di, [arr_dup2D]     ; Завантаження відн. адреси приймача
movcx, 100h             ; Визначення кількості повторень
rep         movsb       ;Завантаження
```

У цьому прикладі команда movsb використана із префіксом повторення rep (repeat – повторювати), що змушує процесор виконати команду movsb кількість разів, відповідно до змісту регістру CX. Ми бачимо, що перед використанням команди movsb треба виконати цілу низку попередніх дій: помістити в регістри DS і ES сегментні адреси джерела й приймача, а в регістри SI й DI ефективну адресу, тобто зміщення у регістрі; за допомогою команди LD (clear direction – скинути напрямок) скинути прапорець процесора DF; у регістр CX записати кількість байт для пересилання. Після цього одна команда movsb із префіксом повторення rep виконує операцію переписування. Кількість переписаних у такий спосіб даних може дорівнювати 32 Кбайт, якщо джерело і приймач перебувають в одному сегменті.

Як відомо, для зручності роботи з більшою кількістю різнорідних файлів у DOS використовується деревоподібна структура каталогів. Каталог – це файл, у якому є перелік всіх підкаталогів наступного рівня й файлів, що входять у цей каталог. Кожному підкаталогу або файлу в каталозі приділяється один елемент розміром 32 байти, у який DOS вносить інформацію про файл, а саме: ім’я, початкову адресу на диску (номер кластера), дату й час створення файлу, довжину файлу у байтах, а також набір характеристик файлу – атрибутів. Крім цього, кожний каталог містить інформацію про себе й батьківський каталог. У разі створення нового файлу МПС під керівництвом DOS «відшукує» на диску вільне місце й призначає його новому файлу. Хоча мінімальною частиною інформації, переданої контролером диска є сектор, і переривання BIOS працюють саме із секторами, файлова система призначає місце на диску цілими кластерами. Розмір кластера на дискеті становить 12 Кбайт. У кластер можуть входити 4–8 секторів. Таким чином мінімальний фізичний розмір файлу становить один кластер.

В елементі каталогу вказується не фізична, а логічна довжина файлу, тобто обсяг даних, що записано у ньому вимірюється у байтах. Робота з файлами передбачає використання дескрипторів (файлових індексів). Це ідентифікатор на рівні операційної системі, для обслуговування файлів, що відкриті.

Стандартна процедура читання-запису файлу в загальному випадку має таку послідовність виконання:

– виклик переривання для відкриття або створення файлу;

– робота з файлом, запис до файлу або читання з файлу;

– закриття файлу і звільнення всіх ресурсів, що були пов’язані з ним.

Переривання DOS 3Ch дозволяє відкрити або створити новий файл для його роботи. Якщо функція 3Ch виявляє, що на диску вже є файл із зазначеним ім’ям, вона фактично знищує його й створює новий з таким самим ім’ям.

Переривання DOS 40h дозволяє записувати дані на будь-який пристрій, зокрема й у файл на диску. Конкретний приймач даних задається його дескриптором. Необхідно зазначити, що під час запису у файл і під час виведення інформації на будь-який пристрій, у приймач даних надходять лише ті дані, які зазначені у програмі. Жодні службові коди, наприклад ознаки кінця файлу, не записуються. Нижче наведено код, що демонструє варіант виконання лабораторної роботи.

```
            TITLE ЛР_5_1
;-------------------------------------------------------------------------
;ЛР  № 5-1 АК
;-------------------------------------------------------------------------
;
; Завдання:
; ВНЗ:                КПІ ім. Ігоря Сікорського
; Факультет:    ФІОТ
; Курс:    1
; Група:       __ __ __
;-------------------------------------------------------------------------
; Автор:        __ __ __
; Дата:         __/__/__
;-------------------------------
;---------------I. ЗАГОЛОВОК ПРОГРАМИ-----------------------------------
IDEAL
MODEL SMALL
```

```
STACK 512
;         II. МАКРОСИ
; 2.2. Складний макрос для ініціалізації
MACRO M_Init                ;Початок макросу
mov    ax, @data     ; ax <- @data
mov    ds, ax ; ds <- ax
mov    es, ax ; es <- ax
ENDM M_Init; Кінець макросу

;         III. ПОЧАТОК СЕГМЕНТА ДАНИХ
DATASEG
; Оголошення двовимірного експериментального масиву 16x16
array2Db db 7, 8 ,7 ,8 ,7 ,8 ,7 ,8 ,7 ,8 ,7 ,8 ,7 ,8 ,7 ,8
db 7, 7 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,7
db 7, 0 ,7 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,7
db 7, 0 ,0 ,7 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,7
db 7, 0 ,0 ,0 ,7 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,7
db 7, 0 ,0 ,0 ,0 ,7 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,7
db 7, 0 ,0 ,0 ,0 ,0 ,7 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,7
db 7, 0 ,0 ,0 ,0 ,0 ,0 ,7 ,0 ,0 ,0 ,0 ,0 ,0 ,7
db 7, 0 ,0 ,0 ,0 ,0 ,0 ,0 ,7 ,0 ,0 ,0 ,0 ,0 ,7
db 7, 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,7 ,0 ,0 ,0 ,0 ,7
db 7, 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,7 ,0 ,0 ,0 ,7
db 7, 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,7 ,0 ,0 ,7
db 7, 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,7 ,0 ,7
db 7, 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,7 ,7
db 7, 8 ,7 ,8 ,7 ,8 ,7 ,8 ,7 ,8 ,7 ,8 ,7 ,8 ,7 ,7

; Для вирівнювання у дампі
arr_def1 dw 3, 0 , 0 , 0 , 0 , 0 , 0 , 0
; Аналог двовимірного масиву
arr_dup2D db 0100h DUP(4)
; Для вирівнювання у дампі
arr_def11 dw 3, 0 , 0 , 0 , 0 , 0 , 0 , 0
arr_rnd1 db 2, 3, 0 , 2, 1, 9, 7, 2
variant db 3
; Для вирівнювання у дампі
arr_def2 dw 3, 0 , 0 , 0 , 0 , 0 , 0 , 0
no_ziro db "Variable is NOT ziro", 10, 13, '$'
is_ziro db "Variable IS ziro", 10, 13, '$'
out_of db "out OF constr", 10, 13, '$'
arr_def6 dw 3, 0 , 0 , 0
exCode db 0

;----------------------VI. ПОЧАТОК СЕГМЕНТА КОДУ-----------------
CODESEG
Start:
M_Init
;------ Stage 1. Write to array2Db[3,3] the arr_rnd1 [0]. Example of first part of Lab 2-
```

```
mov si, 0       ; Index for 2D of array2Db, in gorizontal
mov di, 0       ; Index for 1D of arr_rnd1, in gorizontal
mov dl, [arr_rnd1+di]          ;DL <- arr_rnd1[0]
mov al, 10h
mov dh, [variant]
mul dh          ; AX<-(variant*10h)
mov bx, ax      ; Index for 2D of array2Db, in vertical
mov [array2Db+bx+si+3], dl

;---- Stage 2. Cycle 1 to horisontal. Example of first part of Lab 2-------------------------
mov cx, 8       ; Counter for gorizontal cycle
mov si, 0       ; Index for 2D of array2Db, in gorizontal
mov di, 0       ; Index for 1D of arr_rnd1, in gorizontal
horisontal:
mov dl, [arr_rnd1+di]          ; DL <- arr_rnd1[0]
mov al, 10h
mov dh, [variant]
mul dh          ; AX<-(variant*10h)
mov bx, ax      ; Index for 2D of array2Db, in vertical
inc si   ; Inc for 2D of array2Db, in gorizontal
inc di   ; Inc for 1D of arr_rnd1, in gorizontal
mov [array2Db+bx+si+3], dl
loop horisontal

;----------- Stage 3. The refactoring. Example of first part of Lab 2-------------------------
; we no need for di index!!!
mov cx, 8
mov si, 0       ; Index for 2D of [array2Db] and [arr_rnd1], in gorizontal

horisontal_1:
mov dl, [arr_rnd1+si]          ; DL <- arr_rnd1[0]
mov al, 10h
mov dh, 5       ; For testing refactoring
mul dh          ; AX<-(variant*10h)
mov bx, ax              ; Index for 2D of array2Db, in vertical
inc si   ; inc for 2D of array2Db, in gorizontal
mov [array2Db+bx+si+3], dl
loop horisontal_1
;----------- Stage 4. The gorisontal and vertikal.Example of first part of Lab 2------------
mov dh, [variant]       ; Coutrer of cicle vertical init
vertical_1:
cmp dh, 11      ; The 11 is vertical coord of left bottom of array 8x8
jz main_exit    ; !!! MAIN EXIT
mov al, 10h     ; It is simple mult for vertical coord of strings
mul dh          ; AX<-(variant*10h), execute of mult, result in AX
mov bx, ax      ; BX<-AX Index for [array2Db], in vertical coord
mov cx, 8       ; Counter of cicle gorizontal init
mov si, 0       ; Ziro to index of [array2Db] and [arr_rnd1] in gorizontal
horisontal_2:
mov dl, [arr_rnd1+si]          ; DL <- [arr_rnd1[si]]
inc si   ; inc index of [array2Db] in gorizontal
```

```
mov [array2Db+bx+si+3], dl ;si and 3 are connect to gorizontal coord in 8x8
loop horisontal_2

inc dh ; inc coord vertical
jmp vertical_1          ; come back
main_exit:
;------------------Code organitation with CMP. Analog IF-------------------------
mov cx, 0
cmp cx, 0
jz next_z1
; Code for CX NOT ziro
mov ah, 09h
lea dx, [no_ziro]
int 21h
jmp next_2 ; go to end of constuct IF
next_z1:
; Code for CX IZ ziro
mov ah, 09h
lea dx, [is_ziro]
int 21h
next_2:
mov ah, 09h
lea dx, [out_of]
int 21h
mov ax, 0
;--------------------Code organitation with TEST. Analog IF-----------------
test ax, 10000000b
jne next_z2
; Code for bit of AX IS 1
mov ah, 09h
lea dx, [no_ziro]
int 21h
jmp next_1
next_z2:
; Code for bit of AX IS 0
mov ah, 09h
lea dx, [is_ziro]
int 21h
next_1:
mov ah, 09h
lea dx, [out_of]
int 21h

; Запис великих масивів без організації циклів
; Робота з рядками. Джерело (sourse) у DS:SI, приймач у ES:DI

cld      ; Обнулення прапорця напрямку, прямий
lea si, [array2Db]     ; Завантаження відн. адреси джерела
lea di, [arr_dup2D] ; Завантаження відн. адреси приймача
mov cx, 100h; Визначення кількості повторень
rep      movsb; Завантаження
```

```
; Типова організація циклу, індексна адреса
mov al, 03h
mov cx, 100h ; Заповнення усього масиву
mov si, 0
ptr_1:
mov [ds:[si]], al
inc si
loop ptr_1

; Індексна адресація через ідентифікатор масиву
mov al, 05h
mov cx, 100h ; Заповнення усього масиву
mov si, 0
ptr_2:
mov [array2Db+si], al
inc si
loop ptr_2

Exit:
mov ah,4ch
mov al,[exCode]
int 21h
end Start
```

Отже, для зміни лінійної послідовності виконання операторів і переходу на інші частини програмного коду для організації циклічних операцій і переходу в різні місця коду використовуються команди загального призначення (general-purpose instructions) Асемблера. Вони розділяються на кілька груп: команди переміщення (пересилання, передачі) даних, команди арифметики (додавання, вирахування, множення й ділення) з цілими числами, команди логічних операцій, команди передачі керування (умовних і безумовних переходів, виклик процедур). Для пересилання великих обсягів інформації використовують команди строкових операцій.

## *Перелік запитань для підготовки до лабораторної роботи*

1. Що таке порти введення/виведення, як забезпечується програмний доступ до них у мікропроцесорі 8086?

2. Яку кількість портів введення/виведення підтримує мікропроцесор 8086? Як використати Асемблер для управління зовнішніми пристроями через порти введення/виведення.

3. Опишіть способи використання масочних операцій з командами IN та OUT для управління окремими бітами.

4. Які інтерфейси комп'ютерної систем актуальні нині? Опишіть їх призначення і назву.

5. Які команди Асемблера 8086 використовуються для програмування процедур?

6. Які передаються параметри процедур Асемблера 8086?

7. Як реалізуються цикли у програмах на Асемблері Intel 8086? Поясніть на прикладі коду.

8. Як відбувається програмний доступ до периферійних пристроїв комп'ютера?

# ЛАБОРАТОРНА РОБОТА 6
# СИСТЕМА ОБРОБЛЕННЯ ПЕРЕРИВАНЬ
# В АРХІТЕКТУРІ AMD64 (Intel® 64) У REAL MODE

Мета лабораторної роботи полягає у набутті знань, умінь та навичок для розробки власного програмного переривання на Асемблері для мікропроцесора архітектури AMD64 (Intel® 64) у реальному режимі. Під час використання Асемблера як мови програмування застосовуються знання архітектури мікропроцесора персонального комп'ютера.

## *Завдання на лабораторну роботу* 6

1. Написати з використанням Асемблера процедуру, що виводить на консоль номер групи і прізвище авторів.

2. Написати власне програмне переривання. Вектор переривань, відповідає номеру, що дорівнює 50 та номеру робочої групи.

Для цього переналаштувати процедуру, що розроблена у п. 1 лабораторної роботи на нову процедуру оброблення переривань. Таким чином функціональність переривання буде відповідати процедурі, що отримана у п. 1.

3. Провести тестовий запуск програми. У програмі має бути запуск:

– розробленого переривання до переналаштування на нього функції п. 1;

– розробленого переривання після переналаштування на нього функції п. 1;

– розробленого переривання після повернення системи у вихідний стан.

4. Зробити скриншоти екранів під час роботи програми.

5. Користуючись TD, під час покрокового виконання програми, зробити скриншоти екранів (вікна турбодебагеру), під час переходу на нову функцію обробки переривання. На «фотографії екранів» має бути зміст векторів переривань до переналаштування функції обробки переривань, після переналаштування і після повернення у вихідний стан.

## *Програма проведення експерименту лабораторної роботи 6*

1. Створення вихідного коду. Користуючись одним з текстових редакторів, створити файл вихідного коду лабораторної роботи l6_gr1.asm.

Зберегти його у робочому каталозі, що містить програмну інфраструктуру Асемблера. Записати до сегмента даних вихідний масив і масив з ініціалами студентів робочої групи.

Доробити вихідний код відповідно до поданого завдання.

2. В операційній системі Windows 10 або у сучасній POSIX операційній використовувати віртуальну машину, наприклад DosBox. Запустити цю програму віртуалізації і, використовуючи консоль DosBox, перейти до робочого каталогу.

3. Далі за допомогою програми TASM.EXE провести асемблювання вихідного коду. Для асемблювання вихідного коду у командному рядку виконується така команда:

*tasm.exe /l /zil6_gr1.asm*

Докладний опис команди подано далі у теоретичних відомостях до ЛР 6.

4. Компонування або лінкування. У командному рядку виконується така команда: *tlink.exe/v l6_gr1.obj.*

Докладний опис команди подано далі у теоретичних відомостях до ЛР 6.

5. Перевірити правильність роботи програми на етапі виконання. Переконатися у правильному переналаштуванні функції оброблення переривання і повернення системи у вихідний стан.

У разі появи помилок на етапі асемблювання або лінкування внести зміни до вихідного коду і повторити етапи 3, 4.

6. Після успішного виконання п. 3, 4 провести трасування програми. Для цього необхідно виконати у командному рядку у робочому каталозі таку команду:

*td.exe l6_gr1.exe.*

7. У процесі покрокового виконання програми зробити завдання щодо документації змісту векторів переривання. Зробити висновки.

## *Теоретичні відомості для ЛР 6*

Система переривань є у будь-якій мікропроцесорній системі. Система переривань реалізується програмно-апаратним способом і передбачає джерело переривань, що може бути програмним чи апаратним.

Система переривання дозволяє мікропроцесорній системі у разі виникнення певного класу подій тимчасово перервати виконання поточної програми, передавати управління на функцію обробки переривання. Після завершення останньої перервана програма має бути відновлена з того місця, де вона була перервана.

Кожне переривання викликається сигналом на запит переривання. Залежно від характеру події або джерела сигналу на запит переривання, переривання поділяють на апаратні (зовнішні, внутрішні) і програмні.

Джерелами зовнішніх апаратних переривань можуть бути пристрої периферії ПК, системи управління дисками, нестандартні пристрої. Внутрішні переривання формуються всередині процесора, схемами управління. Прикладом внутрішнього переривання може бути спроба виконання арифметичної дії ділення на 0, порушення рівнів захисту під час спроби виконати команду тощо.

Джерелом програмних переривань є звернення програми користувача до операційної системи. Прикладами є переривання DOS, BIOS.

Процес обробки будь-якого переривання можна розділити на декілька основних етапів. Після початку виконання переривання здійснюється наступна послідовність, що може дещо відрізнятися у різних системах:

1. Встановлення заборони на прийом запитів переривання.

2. Збереження всієї інформації перерваної програми, що необхідна для відновлення виконання цієї програми (контексту програми) після завершення роботи функції оброблення переривання.

3. Зняття заборони на прийом запитів переривання.

4. Ідентифікація джерела запиту переривання його номеру, визначення потрібної функції обробника переривання та її запуск.

5. Завершення функції оброблення переривання.

6. Установка заборони на прийом запитів переривання.

7. Відновлення контексту перерваної програми (повернення до стану на час виникнення переривання).

8. Зняття заборони на прийом запитів переривання.

9. Повернення до виконання перерваної програми.

Мікропроцесори архітектури AMD64 (Intel® 64) залежно від основних режимів роботи [1–2], режим Long Mode, що призначений для 64-розрядних

ОС і режим Legacy Mode (режим забезпечення сумісності), по-різному здійснюють обробку переривання. У режимі Legacy Mode, залежно від базових режимів роботи мікропроцесора (захищений режим роботи (Protected mode), віртуальний 8086 режим роботи (Virtual-8086 mode), реальний режим роботи (Real mode)), також по-різному обробляють переривання.

Реальний режим роботи (Real mode) має найбільш простий механізм обробки переривань і найбільш близький до архітектури I8086. Саме на основі цього режиму у ЛР розглядається механізми роботи переривання.

Приклад програми, що керує системою переривань, показано у прикладі 6-2.

```
TITLE ЛР_6_2  COM1
;-------------------------------------------------------------------------
;ЛР  № 6-2 АК
;-------------------------------------------------------------------------
;
; Завдання:
; ВНЗ:                КПІ ім. Ігоря Сікорського
; Факультет:    ФІОТ
; Курс:    1
; Група:       __ __ __
;-------------------------------------------------------------------------
; Автор:
; Дата:         __/__/__
;-------------------------------
IDEAL

MACRO M_Exit
mov         ah, 04Ch
int         21h
ENDM

MACRO M_Init
mov         ax, @data
mov         ds, ax
mov         es, ax
ENDM

MODEL small
STACK 256

DATASEG

old_offsetDW ? ;  зміщення переривання
old_seg   DW ? ; сегмент переривання
mesg db "Group number:",9,"1",10
db 13,10,13,"Members:",10,13,10,13
```

```
db "Test1",10,13
db "Test1",10,13
db "Test1",10,13,'$'

team = 1 ;
custom_int dw ? ; Змінна для номера переривання, яке перевизначаємо

CODESEG
Start:
M_Init
mov [custom_int], 51 ; номер необхідного переривання
mov     di, [custom_int]   ; Вхідний параметр для процедури
call GIntV

mov     [old_offset], bx ; Запам'ятовуємо зміщення обробника
mov     [old_seg], es ; Запам'ятовуємо сегмент обробника

; Вхідні параметри для процедури SIntV
mov di, [custom_int]
mov dx, offset int_custom ; У DX записуємо зміщення нашої кастомної процедури
коду
mov ax, seg int_custom ; Записуємо логічну адресу сегмента коду
mov es, ax
call SintV

; -------- Виклик
int 51

; ---- Повернення вектора переривання до початкового стану---
; Вхідні параметри для процедури SIntV
mov     di, [custom_int]
mov     dx, [old_offset] ; Записуємо зміщення,
mov     ax, [old_seg] ; Записуємо сегмент, що був спочатку в
mov     es, ax
call SIntV
xor     al, al
M_Exit


;-------------------------------------------------------------------------
; Призначення: Установка на номер вектора нової функції обробника.
; Функціонально: до пам'яті, що відповідає вектору, заносимо ефективну адресу і
адресу сегмента нового обробника переривань
; На вхід:DI - номер переривання, де буде нова процедура
;  DX - ефективна адреса нового обробника
;  ES - адреса сегмента нового обробника
; На вихід: ---
;-------------------------------------------------------------------------
; Автор:
; Дата:       __/__/__
;---------------------------------

PROC SIntV
cli ; Заборона апаратних переривань
```

```
; Поміщаємо в стек регістри, значення яких хочемо залишити недоторканним
push ax
push di
push ds

xor         ax, ax ; AX <- 0
mov ds, ax    ; DS <- 0
shl         di, 2 ; DI <- DI * 4
mov         [ds:[di]], dx ; Перезаписуємо зміщення у векторі переривання
mov         [ds:[di + 2]], es ; Перезаписуємо сегмент у векторі переривання

; Відновлюємо попередні значення регістрів зі стека
pop         ds
pop         di
pop         ax

sti ; Дозвіл апаратних переривань
ret
ENDP


;--------------------------------------------------------------------------
; Процедура - кастомний обробник для переривання.
; Вхід: --
; Вихід: --
;--------------------------------------------------------------------------
; Автор:
; Дата:       __/__/__
;-------------------------------
PROC int_custom

mov ax, 03h
int 10h ; Очищення екрану
mov ah, 09h ; Ф-я виводу рядка на екран
mov dx, offset mesg ; DX <- зміщення змінної mesg
int 21h ; Переривання DOS 21h
; Код закінчення апаратного переривання
mov         al, 20h
out         20h, al
iret        ; Вихід з процедури обробника переривання
ENDP

;--------------------------------------------------------------------------
; Призначення: Отримання логічної адреси процедури (функції) обробки
переривання за номером вектора переривання
; Вхід:DI <- номер вектора переривання
; Вихід:BX,ES <- Ефективна адреса та сегмент процедури (функції) обробки
переривання
;--------------------------------------------------------------------------
; Автор:
; Дата:       __/__/__
```

```
;--------------------------------

PROC GIntV
; Поміщаємо в стек регістри, значення яких хочемо залишити недоторканним
push ax
push di

xor     ax, ax ; AX <- 0
mov     es, ax ; ES <- 0
shl     di, 2 ; DI <- DI * 4
mov     bx, [es:[di]] ; В BX записуємо
mov     ax, [es:[di + 2]] ; В AX записуємо
mov     es, ax
pop     di
pop     ax
ret
ENDP
END Start
```

Відповідно переривання розподіляються на програмні й апаратні. Переривання може бути синхронним, що викликається програмно. У цьому випадку переривання реалізуються через операційну систему або у BIOS. Переривання може бути асинхронним, джерелом якого є апаратна подія, клавіатура, «мишка», послідовний інтерфейс тощо. Основна відмінність асинхронного переривання полягає у його непередбаченості (випадковості) у часі.

У архітектурі I8086 – дозвіл переривань визначається станом прапорця IF. Якщо він піднятий до 1, то переривання дозволені, інакше апаратні переривання не дозволені. Є дві команди Асемблера, що керують цим прапорцем. Команда Асемблера cli обнуляє прапорець IF, забороняючи переривання, команда sti встановлює прапорець в 1.

В архітектурі I8086 векторний механізм переривання. Переривання називається векторним, якщо кожному джерелу переривання відповідає вектор переривання. Вектор переривання містить логічну адресу відповідної функції обробки переривання. Програмно-апаратний механізм переривання дозволяє автоматично передати управління функції обробки переривання.

В архітектурі I8086 вектор переривання зберігає значення CS:IP, тобто логічну адресу процедури (функції) оброблення переривання. Ця логічна адреса містить 4 байти (32 біти). Вектори апаратних переривань звичайно зберігаються у початковій ділянці оперативної пам’яті.

У першому слові (2 байти) зберігається значення ефективної адреси початку функції оброблення переривання (IP), у другому – адреса сегмента – CS. Молодші 1024 байти ОП містять вектори переривань, зокрема апаратні. Таким чином зарезервовано і використовується 256 векторів переривань. У цілому утворюється таблиця векторів переривань.

Вектор номером 0 починається з адреси 0000:0000 і закінчується адресою 0000:0003, вектор 1 починається з адреси 0000:0004 і закінчується 0000:0007 тощо.

У мікропроцесорі 8086 є два типи апаратних переривань:

(A) NMI (Non Maskable Interrupt) немасковане апаратне переривання, що не можна вимкнути, після його виконання генерується переривання TYPE 2;

(B) INTR масковані переривання, що можуть бути відмінені програмно або затримані, джерелом є периферійні пристрої, значення IP та CS можуть бути змінені залежно від типу переривання.

Мікропроцесор I8086 також має 256 програмних переривань. Інструкції Асемблера для виклику програмного переривання мають формат INT (номер). Номер переривання має значення від 00 до FF.

Процедура (функція) оброблення переривання звичайно закінчується командою IRET (повернення з переривання). Ця команда повертає МПС на той рядок коду, у якому була програма до початку виникнення переривання. Для цього зі стека повертаються і завантажуються вихідні адреси до пари регістрів CS:IP. Повертається зі стека регістр прапорців вихідного стану програми.

Під час роботи функції обробки переривання можуть виникати інші апаратні або програмні переривання. Для упорядкування процесу обробки переривань є система пріоритетів.

Система пріоритетів дозволяє вибрати те переривання, пріоритет якого вище, зупиняючи функцію обробки переривання з нижчим пріоритетом. Аналогічно організовані програмні переривання. Відмінність їх полягає в тому, що вони не викликаються апаратною частиною МПС. Їх джерелом є програмні команди. Найбільш відомими є переривання DOS, наприклад INT21h. Це переривання було використано у ЛР 1, ЛР 2. Незалежну від ОС групу програмних переривань сумісних комп’ютерів утворюють переривання BIOS, наприклад INT09h.

Функції (процедури) оброблення переривань можуть бути переписані й переназначені за потребою програмістом. Для архітектури I8086 вико-

ристовували мікросхему контролера переривань Intel 8259. Нині це реалізується у інший спосіб. Мікросхема Intel 8259 для упорядкування роботи має 8 апаратних входів і 8 рівнів пріоритетів апаратних переривань, що нумеруються IRQ0–IRQ7. Дві поєднані мікросхеми Intel 8259 утворюють розширення діапазону переривань IRQ0–IRQ15. Максимальний пріоритет апаратного переривання відповідає рівню 0. Звичайно, кожному апаратному входу (IRQ0–IRQ7) відповідає свій вектор переривань у таблиці переривань Далі подано приклади векторів та їх пріоритети:

IRQ 0 таймер;

IRQ 1 клавіатура;

IRQ 2 канал введення/виведення;

IRQ 8 годинник реального часу (тільки AT); IRQ 9 програмно переводяться в IRQ2 (AT); IRQ 10 резерв;

IRQ 11 резерв;

IRQ 12 резерв;

IRQ 3 COM1 (COM2 для AT);

IRQ 4 COM2 (модем для PCjr, COM1 для AT).

Відомо декілька причин написання власного обробника переривань: для зручності та зменшення обсягу коду; для реалізації функції оброблення нового апаратного переривання нестандартного обладнання; доробка процедури оброблення переривання, що запрограмована в ОС.

Отже, механізми обробки переривань у мікропроцесорах архітектури AMD64 (Intel® 64) залежать від режимів роботи. У реальному режимі роботи (Real mode) найбільш простий механізм обробки переривань і найбільш близький до архітектури I8086. В архітектурі I8086 векторний механізм переривання містить два типи апаратних переривань:

(A) NMI (Non Maskable Interrupt) немасковане апаратне переривання, що не можна вимкнути;

(B) INTR масковані переривання, що можуть бути відмінені програмно або затримані. Мікропроцесор I8086 також має 256 програмних переривань. Функції (процедури) оброблення переривань можуть бути переписані й переназначені за потребою програмістом.

## *Перелік запитань для підготовки до лабораторної роботи*

1. Рішення яких проблем дозволяє вирішити маскування переривань?

2. Опишіть загальну схему обробки переривання?

3. Як забезпечується відновлення обчислень після обробки переривання?

4. Для чого в обробнику переривань передбачено тимчасову заборону на прийом запитів переривання? Коли він вводиться і знімається?

5. Що таке вектор переривання і як він пов’язаний з функцією обробки переривання?

6. Назвіть призначення та склад системної плати.

7. Яку функцію виконує Northern Bridge (Північний міст), Southern Bridge (Південний міст)? Як змінений системний дизайн у сучасних комп’ютерах.

8. Опишіть механізм конвеєрної обробки команд у процесорі.

9. Опишіть принцип суперскалярної архітектури процесора.

10. Дайте характеристику периферійної шини USB.

11. Опишіть типи, призначення регістрів і організацію пам’яті мікропроцесора 8086.

12. Перерахуйте особливі функції регістрів загального призначення архітектури Intel 8086.

13. Перерахуйте сегментні регістри архітектури Intel 8086. Для чого призначені регістри, які їх особливості?

14. Як здійснюється сегментація пам’яті 8086, що дозволяє покращити цей механізм?

# ЛАБОРАТОРНА РОБОТА 7
# ПРОЦЕДУРИ АСЕМБЛЕРА І ПОРТИ ВВЕДЕННЯ/ВИВЕДЕННЯ АРХІТЕКТУРИ AMD64

Мета лабораторної роботи полягає у набутті знань, умінь та навичок розроблення ПЗ з текстовим інтерфейсом користувача на Асемблері для мікропроцесора архітектури I8086. Крім того, розглядаються питання адресації периферії через порти введення/виведення.

Під час використання Асемблера, як мови програмування застосовуються знання архітектури мікропроцесора персонального комп'ютера і процедур Асемблера.

## *Завдання на лабораторну роботу 7*

1. Написати програму, що реалізує текстовий інтерфейс і підпрограми, що працюють через нього. У табл. 7.1 показано варіанти для сполучення букв текстового інтерфейсу.

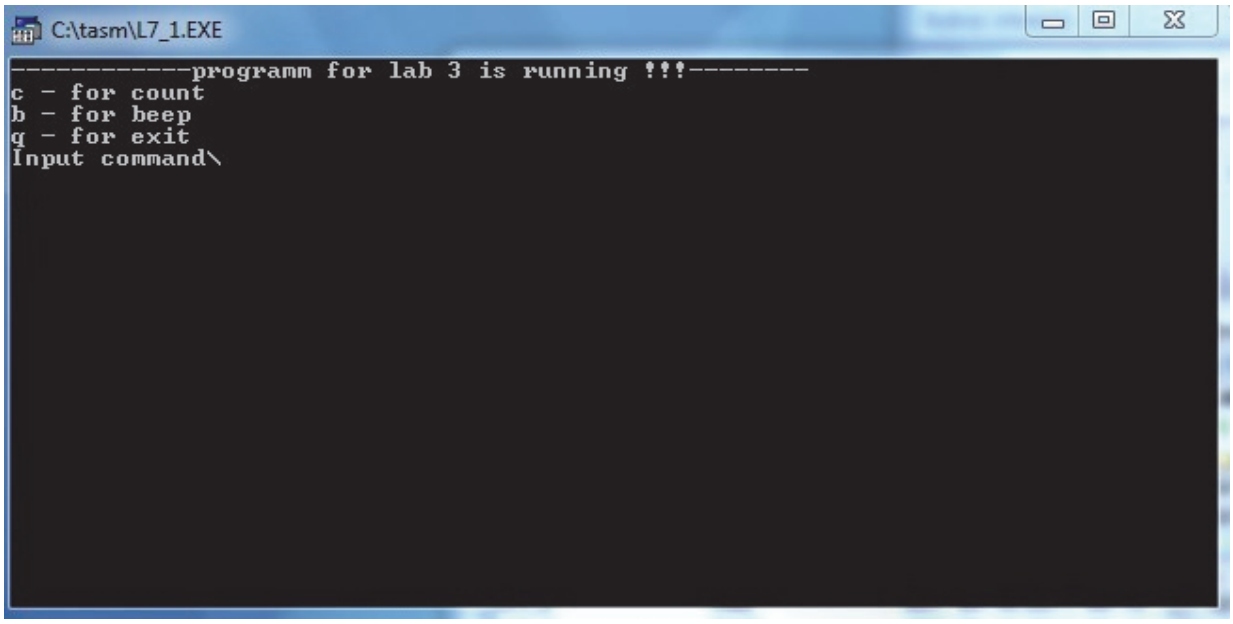



*Рис. 7.1*. Інтерфейс програми

2. Після розробки програми інтерфейс користувача має приблизно такий вигляд, як показано на рис. 7.1.

3. Сполучення букв у інтерфейсі користувача потрібно змінити відповідно до варіантів завдань, що подано в табл. 7.1.

*Таблиця 7.1*

**Сполучення букв для розроблення інтерфейсу користувача**

| Функції | Номер варіанта | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
| 1. Літера для виклику функції обчислення виразу | q | r | y | i | a | G | l | c | n | . | 2 | 5 | 8 | e | t | u | i | o |
| 2. Літера для виклику функції включення звуку (тривалість звучання, с) | W 1 | T 2 | U 3 | O 4 | S 5 | H 4 | Z 3 | V 2 | M 1 | / 2 | 3 4 | 6 7 | 9 8 | D 9 | H 1 | J 3 | K 4 | L 5 |
| 3. Літера для виходу з програми | e | y | i | p | d | k | x | b | , | 1 | 4 | 7 | 0 | c | n | m | , | . |
| 4. Літера для пошуку найбільшого значення *(парний варіант)* | | a | | b | | c | | d | | f | | g | | h | | r | | o |
| 5. Літера для пошуку найменшого значення *(непарний варіант)* | q | | w | | e | | t | | r | | y | | u | | o | | p | |
| 6. Частота звуку, кГц | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 1 | 2 | 7 | 8 | 9 | 10 | 11 |

4. Написати процедуру для розрахунку виразу, що викликається, під час натискання відповідної клавіші на клавіатурі. Процедуру можна реалізувати з цілочисловими значеннями і зі знаковими числами. Студент повинен читати і пояснювати свій розроблений код програми.

*1.* $((a_1+a_2)*a_3/a_4+a_5)$ $a_1=-7, a_2=3, a_3=2, a_4=4, a_5=1$

*2.* $((a_1+a_2)*a_3/a_4+a_5)$ $a_1=-7, a_2=3, a_3=2, a_4=4, a_5=2$

*3.* $((a_1+a_2)*a_3/a_4+a_5)$ $a_1=-7, a_2=3, a_3=2, a_4=4, a_5=3$

| | |
|---|---|
| *4. ((a1-a2)*a3*a4+a5)* | a1=-1, a2=1, a3=2, a4=2, a5=3 |
| *5. ((a1-a2)*a3*a4+a5)* | a1=-1, a2=2, a3=1, a4=2, a5=3 |
| *6. ((a1-a2)*a3*a4+a5)* | a1=-1, a2=1, a3=1, a4=2, a5=3 |
| *7. (((a1-a2)+a3)/a4*a5)* | a1=-2, a2=3, a3=1, a4=2, a5=3 |
| *8. (((a1-a2)+a3)/a4*a5)* | a1=-2, a2=3, a3=1, a4=2, a5=3 |
| *9. (((a1-a2)+a3)/a4*a5)* | a1=-2, a2=3, a3=1, a4=2, a5=3 |
| *10. (a1-a2*a3/a4+a5)* | a1=-6, a2=3, a3=2, a4=2, a5=1 |
| *11. (a1-a2*a3/a4+a5)* | a1=-6, a2=3, a3=2, a4=2, a5=1 |
| *12. (a1-a2*a3/a4+a5)* | a1=-6, a2=3, a3=2, a4=2, a5=1 |
| *13. (((a1-a2)/a3 - a4)*a5)* | a1=-3, a2=3, a3=2, a4=1, a5=2 |
| *14. (((a1-a2)/a3 - a4)*a5)* | a1=-3, a2=3, a3=2, a4=1, a5=2 |
| *15. (((a1-a2)/a3 - a4)*a5)* | a1=-3, a2=3, a3=2, a4=1, a5=2 |
| *16. (((a1+a2)/a3 + a4) - a5)* | a1=-8, a2=4, a3=2, a4=1, a5=1 |

Сполучення букв у інтерфейсі користувача потрібно змінити відповідно до варіантів завдань, що подані у табл. 7.1.

6. Для 4–5 рядка у табл. 7.1 потрібно написати процедуру, що робить сортування тестового масиву. Масив визначити у сегменті даних, розмірність і тип даних за вибором студента. Можна скористатися масивами з ЛР 2–5.

7. Для крайнього рядка табл. 7.1 написати процедуру, що генерує звук частотою відповідної до варіанта. Літеру до графічного інтерфейсу підібрати самостійно.

## *Програма проведення експерименту лабораторної роботи 7*

1. Створення вихідного коду. Користуючись одним з текстових редакторів, створити файл вихідного коду лабораторної роботи l7_gr1.asm. Зберегти його у робочому каталозі, що містить програмну інфраструктуру Асемблера. Записати до сегмента даних вихідний масив і масив з ініціалами студентів робочої групи.

Доробити вихідний код відповідно до поданого завдання.

2. В операційній системі Windows 10 або у сучасній POSIX операційній використовувати віртуальну машину, наприклад DosBox. Запустити цю програму віртуалізації і, використовуючи консоль DosBox, перейти до робочого каталогу.

3. Далі за допомогою програми TASM.EXE провести асемблювання вихідного коду. Для асемблювання вихідного коду у командному рядку виконується така команда:

*tasm.exe /l /zil7_gr1.asm*

Докладний опис команди подано нижче у теоретичних відомостях до ЛР 7.

4. Компонування або лінкування. У командному рядку виконується така команда:

*tlink.exe/v l7_gr1.obj.*

Докладний опис команди подано нижче у теоретичних відомостях до ЛР 7.

5. У разі появи помилок на етапі асемблювання або лінкування внести зміни до вихідного коду і повторити етапи 3, 4.

6. Після успішного виконання п. 3, 4 провести трасування програми. Для цього необхідно виконати у командному рядку у робочому каталозі таку команду:

*td.exe l7_gr1.exe.*

7. Запустити програму і переконатися у її правильній роботі на етапі виконання. У випадку некоректності зробити виправлення. Зробити скриншоти екранів (фотографування екранів під час роботи) на кожному кроці і додати їх до звіту. Записати результати до звіту.

8. У процесі покрокового виконання за допомогою дослідження дампу пам’яті сегмента даних і сегмента стека переконатися у правильному виконанні завдання ЛР. Зробити висновки щодо результатів роботи програми на Асемблері.

## *Теоретичні відомості для ЛР 7*

Мікропроцесори архітектури AMD64 (Intel® 64) мають окрему систему і адресний простір для управління пристроями введення/виведення.

На рис. 7.1 показано простір введення/виведення мікропроцесорів архітектури AMD64 (Intel® 64). Порти введення/виведення можуть розглядатися як байти, слова, або подвійні слова.

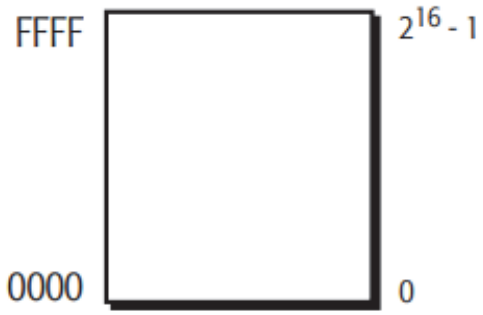



*Рис. 7.1*. Адресний простір портів введення/виведення AMD64 (Intel® 64) [1]

Обмін інформацією з зовнішніми пристроями багато в чому подібний обміну з пам'яттю. Організація взаємодії між центральним процесором і зовнішніми пристроями через порти введення/виведення в мові Асемблер здійснюється за допомогою команд введення/виведення: IN і OUT.

Команда IN (Input operand from port) служить для введення значень з порту.

Синтаксис команди: IN акумулятор, ном_порту.

Команда передає байт або слово з порту введення/виведення в один з регістрів: al або ax і застосовується для прямого керування устаткуванням комп'ютера за допомогою портів. Номер порту задається у вигляді прямої адреси, що міститься в другому байті або команді у вигляді непрямої адреси, яка записана до dx. Безпосереднім значенням можна задати порт із номером у діапазоні 0…255... Під час використання порту з більшим номером застосовується непряма адресація через регістр dx. Розмір даних, що вводяться, визначається довжиною першого операнда (акумулятора) і може бути байтом або словом.

Команда OUT (Output operand to port) служить для виведення значень у порт. Синтаксис команди: OUT ном_порту, акумулятор.

Команда передає байт або слово з регістра al чи ax у порт, номер якого визначається першим операндом, і застосовується для прямого керування устаткуванням комп'ютера за допомогою портів. Номер порту задається першим операндом у вигляді безпосереднього значення або значенням у регістрі dx. Також, як і під час введення, безпосереднім значенням можна задати порт із номером у діапазоні 0…255. Для вказівки порту з великим номером використовується регістр dx. Розмір даних визначається довжиною другого операнда (акумулятора) і може бути байтом або словом.

Розглянемо приклад програмування з використанням команд введення/виведення. Для цього складемо програму керування світлодіодами клавіатури.

Керування клавіатурою здійснюється за допомогою звертання до портів введення/виведення. Порт з адресою 64h використовується для читання регістра стану контролера клавіатури. Під час читання порт повертає наступний байт:

біт 7: помилка парності під час передачі даних із клавіатури;

біт 6: тайм-аут під час прийому;

біт 5: тайм-аут під час передачі;

біт 4: клавіатура закрита ключем;

біт 3: дані, записані в регістр введення, – команда;

біт 2: самотестування закінчено;

біт 1: у буфері введення є дані (для контролера);

біт 0: у буфері виведення є дані (для комп'ютера).

Порт з адресою 60h призначений для запису команд у регістр керування контролера клавіатури. Байт записується в цей порт, якщо біт 1 під час читання порту 64h дорівнює 0 (див. вище) й інтерпретується як команда. Деякі команди можуть складатися з більш ніж одного байта. У цьому випадку варто дочекатися обнуління біта 1 у регістрі стану контролера клавіатури, а потім послати наступний байт.

Підпрограма тестування 1 біта у потру 64 h може мати такий вигляд:

```
PROC  wait_kbd          ;Початок підпрограми

test_kbd:               ;Мітка початку цикла
in    al,   64h         ;Читання стану контролера клавіатури
test  al,   2           ;Перевірка біта готовності (2=00000010b)
jnz  test_kbd           ;Перехід на мітку test_kbd, якщо прапорець Z=0,
                        ;тобто біт готовності дорівнює 1 (буфер зайнятий),
                        ;інакше (Z=1, тобто біт готовності дорівнює 0
                        (буфер; пустий)) – перехід на команду ret
ret                     ;повернення із підпрограми
ENDP  wait_kbd          ;Кінець підпрограми
```

В архітектурі AMD64 (Intel® 64) команда IN передає байт, слово або подвійне слово з порту введення/виведення в регістр AL, AX або EAX. Порт

адреси вказується або 8-бітовим безпосереднім значенням (від 00h до FFh), закодованим в інструкції, або 16-бітове значення, що міститься в регістрі DX (від 0000h до FFFFh). Адресний простір введення/виведення процесора не належить до адресного простору оперативної пам'яті.

Приклад управління портом введення/виведення і формування звуку показано у вихідному коді, що подано далі. У цій програмі використовується доступ для портів введення/виведення таймеру з метою формування відповідного сигналу.

Для формування графічного інтерфейсу користувача запропоновано розглянути вихідний код, що подано нижче. Програма реалізована для Асемблера TASM. У якості спрощеного прототипу для виконання лабораторної роботи пропонується вихідний код, який дає можливість сформувати консольну програму з текстовим інтерфейсом користувача.

```
TITLE ЛР_7_0
;-------------------------------------------------------------------------
;ЛР  № 7-0 АК
;-------------------------------------------------------------------------
;
; Завдання:
; ВНЗ:                КПІ ім. Ігоря Сікорського
; Факультет:    ФІОТ
; Курс:    1
; Група:       __ __ __
;-------------------------------------------------------------------------
; Автор:        __ __ __
; Дата:         __/__/__
;--------------------------------
IDEAL
MODEL   small
STACK   256

DATASEG
        string  db      254     ;string variable def. There is max len ,
        str_len db      0
        db 254 dup ('*')        ; Буфер
        system_message_1   DB "Input somthing\   "  ,'$'
        display_message_0  DB "------------menu bagin--------", 13, 10, '$'
        display_message_1  DB "c - for count",   13, 10,  '$'
        display_message_3  DB "q - for exit",    13, 10,  '$'
        display_message_4  DB "----------programm for lab is END !!! - for exit", 13, 10,
'$'
        display_message_5  DB "------------menu end--------", 13, 10, '$'
        message  DB ?
```

```
        test_message_1  DB "COUNT",        13, 10,  '$'
        test_message_3  DB "EXIT",         13, 10,  '$'

CODESEG

Start:
mov ax, @data
mov ds, ax

Main_cycle:

        call display1

        call input_foo

        cmp ax, 063h ; c ascii =63h
        je Count
        cmp ax, 071h ; q ascii =71h
        je  Exit
        jmp Main_cycle

Count:
        mov dx, offset test_message_1
        call display_foo  ; any foo
        jmp Main_cycle


Exit:
mov dx, offset display_message_4
call display_foo
mov ax,04C00h
int 21h
;-------------------------------------------------
PROC display1
        mov dx, offset display_message_0
        call display_foo
        mov dx, offset display_message_1
        call display_foo
        mov dx, offset display_message_3
        call display_foo
        mov dx, offset system_message_1
        call display_foo
        mov dx, offset display_message_5
        call display_foo
        ret
ENDP display1
;-------------------------------------------------
PROC display_foo; input dx is offset
        mov ah,9
        int 21h
        xor dx, dx
```

```
        ret
ENDP display_foo
;-------------------------------------------------
PROC input_foo              ; input string  out ax
mov ah, 0ah                 ; ah <- 0ah input
mov dx, offset string       ; dx <- offset string
int 21h                     ; call  0ah function DOS int 21h

        xor ax, ax
        mov bx, offset string
        mov ax, [bx+1]
        shr ax, 8
        ret
ENDP input_foo

END Start
```

У головному циклі програми відбувається відображення текстового меню програми. Далі викликається процедура, що виводить до регістру AX код натиснутої клавіші. Після цього у конструкції вибору визначається логіка програми.

Створюємо програму, користуючись вихідним кодом, що подано далі.

```
TITLE ЛР_7_1
;-------------------------------------------------------------------------
;ЛР  № 7-1 АК
;-------------------------------------------------------------------------
;
; Завдання:
; ВНЗ:                КПІ ім. Ігоря Сікорського
; Факультет:    ФІОТ
; Курс:    1
; Група:       __ __ __
;-------------------------------------------------------------------------
; Автор:        __ __ __
; Дата:         __/__/__
;-------------------------------
IDEAL
MODEL   small
STACK   256
DATASEG

string  db      254     ;змінна для строки - string,
str_len db      0       ;
db 254 dup ('*')        ; Буфер заповнюється '*' для ;кращого налаштування
;----- Змінні для виводу системних команд
system_message_1   DB "Input command and press enter\  "  ,'$'
```

```
system_message_2   DB "Program end"       ,'$'
;----- Змінні для виводу команд під час управління програмою
display_message_0  DB "------------programm for lab 3 is running !!!--------", 13, 10, '$'
display_message_1  DB "c - for count",   13, 10,  '$'
display_message_2  DB "b - for beep",    13, 10,  '$'
display_message_3  DB "q - for exit",    13, 10,  '$'
display_message_4  DB "----------programm for lab is END !!! ----------", 13, 10,  '$'
display_message_5  DB "Press any key for beep ----------", 13, 10,  '$'
;--- Змінні що використовувалися під час налаштування програми
message  DB ?
;test_message_1  DB "!!! count DISPLAY",      13, 10,  '$'
;test_message_2  DB "!!! beep DISPLAY",       13, 10,  '$'
;test_message_3  DB "q - for exit",           13, 10,  '$'
;------Константи  для функції звуку
NUMBER_CYCLES  EQU   2000
FREQUENCY      EQU   600
PORT_B         EQU   61H
COMMAND_REG    EQU   43H     ; Адреса командного регістру
CHANNEL_2      EQU   42H       ; Адреса каналу 2
simvol  db ?

CODESEG
Start:
mov ax, @data           ;
mov ds, ax
Main_: ; Основний цикл програми для інтерфейсу користувача
;-------------------------------------------
call display_foo_main
;-------------------------------------------
mov ah, 0ah             ; ah <- 0ah
mov dx, offset string    ; пересилання в dx початку буфера
int 21h
xor ax, ax
mov bx, offset string       ;пересилання в bx початку буфера для
;реалізації адресації зі зміщенням
mov ax, [bx+1]              ;занесення в ax чисельного значення
;символу ASCII, що відповідає
;знаку,
;який введено з клавіатури

shr ax, 8                   ;зсув в  регістрі ax для виконання
;cmp
cmp ax, 063h ; c ascii =63h   ; Вибір відповідної функції
je Count                     ; На лекції 3!!!
cmp ax, 062h ; b ascii =62h
je Beep
cmp ax, 071h ; q ascii =71h
je  Exit
jmp Main_
;-------------------------------------------------------
Count:
```

```
; mov dx, offset test_message_1 ; Закоментовані повідомлення
; у процесі налаштування
; call display_foo              ; тут повинна викликатися
; функція для обчислення
; виразу і виведення
; результату на консоль
; any foo for counte
jmp Main_
;-------------------------------------------------------
Beep:
; any foo for sound             ; виклик функції звуку
mov dx, offset display_message_5
call display_foo
call  SNDF1
jmp Main_
;-------------------------------------------------------
; Стандартний вихід з програми
Exit:
mov dx, offset display_message_4
call display_foo
mov ah,04Ch          ;
int 21h              ;
;-----------------------------------------
; Відображення Інтерфейсу користувача
; На вхід:      ---
;
; На вихід: ---
;-----------------------------------------
PROC display_foo_main
mov ah, 0
mov al, 3
int 10h
mov dx, offset display_message_0
call display_foo
mov dx, offset display_message_1
call display_foo
mov dx, offset display_message_2
call display_foo
mov dx, offset display_message_3
call display_foo
mov dx, offset system_message_1
call display_foo
ret
ENDP display_foo_main

;-----------------------------------------
; Відображення рядка
; На вхід:      dx, offset system_message_1
;
; На вихід: ---
;-----------------------------------------
```

```
PROC display_foo
mov ah,9
int 21h
xor dx, dx
ret
ENDP display_foo

;-----------------------------------------------
; Генерація звуку на сист. динамік
; На вхід:      ---
;
; На вихід: ---
;-----------------------------------------------
PROC  SNDF1

int 16h              ; Зберігає отримане значення з клавіатури  у змінній
mov [simvol],al      ; simvol
cmp [simvol],'e'; Перевірка на відповідність і встановлення прапорця ознаки 0
jz Exit              ;
; Перехід на Exit: у випадку відповідності

;Встановлення частоти 440 гц

;--- дозвіл каналу 2 встановлення порту  В мікросхеми 8255
in   AL,PORT_B    ;Читання
or   AL,3         ;Встановлення двох молодших бітів
out  PORT_B,AL    ;пересилання байта в порт В мікросхеми 8255

;--- встановлення регістрів порту введення/виведення
mov  AL,10110110B ;біти для каналу 2
out  COMMAND_REG,AL  ;байт в порт командний регістр

;--- встановлення лічильника
mov  AX,2705         ;лічильник = 1190000/440
out  CHANNEL_2,AL    ;відправка AL
mov  AL,AH  ;відправка старшого байту в AL
out  CHANNEL_2,AL    ;відправка старшого байту

;--- виклик преривання з клавіатури для зупинки
mov  AH,8    ;номер функції преривання 8
int  21H         ;виклик преривання

;--- виключення звуку
in   AL,PORT_B    ;отримуємо байт з порту В
and  AL,11111100B  ;скидання двох молодших бітів
out  PORT_B,AL    ;пересилання байтів у зворотному напрямку
ret
ENDP SNDF1

END Start
```

Опишемо програму більш детально. На початку роботи програма викликає функцію *display_foo_main*. Вона призначена для виведення повідомлення на консоль для користувача.

Як можна побачити з наведеного вище коду, на першому етапі здійснюється очищення екрана, далі послідовно викликається функція *display_foo*, що дає можливість вивести на екран відповідне повідомлення, передача параметрів у функцію здійснюється через регістр *dx*.

Після виведення повідомлення виконується основний цикл програми.

Фрагмент його коду подано нижче.

```
Main_: ; Основний цикл програми для інтерфейсу користувача
;-------------------------------------------
call display_foo_main
;-------------------------------------------
mov ah, 0ah             ; ah <- 0ah
mov dx, offset string    ; пересилання в dx начала буфера
int 21h
xor ax, ax
mov bx, offset string
mov ax, [bx+1]
shr ax, 8

cmp ax, 063h
je Count
cmp ax, 062h
je Beep
cmp ax, 071h
je  Exit
jmp Main_
;-------------------------------------------------------
Count:
; mov dx, offset test_message_1 ; Закоментовані повідомлення
; у ході налаштування
; call display_foo            ; тут повинна викликатися
; функція для обчислення
; виразу і виведення
; результату на консоль
; any foo for counte
jmp Main_
```

Аналізуючи вихідний код можна зробити висновок, що залежно від введеного з клавіатури знака (c, b або q) викликаються відповідні функції. Під час набору на клавіатурі й натискання клавіші «Enter» – здійснюється розрахунок виразу (в шаблоні програми його немає, його потрібно розробити

відповідно завдання лабораторної роботи). У разі набору b і натискання клавіші «Enter» викликається звук динаміка, а у разі набору q і натискання клавіші «Enter» здійснюється вихід з програми.

Отже, мікропроцесори архітектури AMD64 (Intel® 64) мають окрему систему для управління пристроями введення/виведення і пристроями управління. Ця система має окремий адресний простір для управління пристроями введення/виведення. Він не перетинається з адресним простором оперативної пам'яті. Порти введення/виведення можуть розглядатися як байти, слова, або подвійні слова, але їх основне завдання – це управління пристроями введення/виведення і системами.

Для управління у порти записується певна інформація і є дві команди, що призначені саме для запису до портів (команда OUT) і зчитування інформації з портів (команда IN).

## *Перелік запитань для підготовки до лабораторної роботи*

1. Опишіть типи, призначення регістрів і організацію пам'яті мікропроцесорів архітектури IA-32.

2. Перерахуйте регістри загального призначення мікропроцесорів архітектури IA-32. Для чого призначені регістри, які їх особливості? Розкрийте поняття розрядності регістрів.

3. Перерахуйте сегментні регістри архітектури IA-32. Розкрийте поняття розрядності регістрів. Що зберігається у цих регістрах під час виконання програми.

4. Перерахуйте системні регістри мікропроцесорів архітектури IA-32. Для чого призначені регістри, які їх особливості?

5. Як здійснюється адресація у мікропроцесорах архітектури IA-32 у реальному і захищеному режимах, навіщо розроблений цей механізм?

# ЛАБОРАТОРНА РОБОТА 8
# АРХІТЕКТУРА IA-32(X86) У REAL MODE

Мета лабораторної роботи полягає у набутті знань, умінь та навичок розроблення ПЗ із псевдографічним інтерфейсом користувача на Асемблері для мікропроцесора архітектури I8086. Під час використання Асемблера як мови програмування застосовуються знання архітектури мікропроцесора персонального комп'ютера і процедур Асемблера.

## *Завдання на виконання лабораторної роботи 8*

1. Розробити псевдографічний інтерфейс, що показано на рис. 8.1.
Під час розробки необхідно додати у меню три додаткових кнопки.


*Рис. 8.1.* Меню для вихідного коду файлу STRMENU.asm

2. Під час натискання першої кнопки на екран виводиться номер робочої групи, ім'я учасників групи – латиницею з використанням переривань DOC.

3. Повторити функціональність лабораторної роботи 7, а саме виклик процедури обчислення виразу, виклик звуку і вихід з програми відповідно до варіанту ЛР 7. Далі подано програму, що реалізує інтерфейс поданий у завданні. Під час виконання лабораторної роботи надається можливість реалізувати інтерфейс за власним шаблоном.

### *Програма проведення експерименту лабораторної роботи 8*

1. Створення вихідного коду. Користуючись одним з текстових редакторів, створити файл вихідного коду лабораторної роботи l8_gr1.asm. Зберегти його у робочому каталозі, що містить програмну інфраструктуру Асемблера. Записати до сегмента даних вихідний масив і масив з ініціалами студентів робочої групи.

Доробити вихідний код відповідно до поданого завдання користуючись результатами ЛР 7.

2. В операційній системі Windows 10 або у сучасній POSIX операційній використовувати віртуальну машину, наприклад DosBox. Запустити цю програму віртуалізації і, використовуючи консоль DosBox, перейти до робочого каталогу.

3. Далі за допомогою програми TASM.EXE провести асемблювання вихідного коду. Для асемблювання вихідного коду в командному рядку виконується така команда:

*tasm.exe /l /zil8_gr1.asm*

4. Компонування або лінкування. У командному рядку виконується така команда: *tlink.exe/v l8_gr1.obj.*

5. У разі появи помилок на етапі асемблювання або лінкування внести зміни до вихідного коду і повторити етапи 3, 4.

6. Після успішного виконання п. 3, 4 провести трасування програми. Для цього необхідно виконати у командному рядку у робочому каталозі таку команду:

*td.exe l5_gr1.exe.*

7. Запустити програму. Провести її тестування на рівні виконання. Результати тестування зафіксувати за допомогою скриншотів (фотографування екранів під час роботи програми). Записати результати до звіту. Зробити висновки щодо результатів роботи програми на Асемблері.

У якості методичних рекомендацій далі подано вихідний код, за основу взято результати [5]. Коротко розглянемо роботу програми. Якщо натискаємо кнопку клавіатури «стрілка догори», курсор зсувається вверх, елемент меню виділяється синім кольором, як показано на рисунку.

```
TITLE ЛР_8
;-------------------------------------------------------------------------
;ЛР  № 8
;-------------------------------------------------------------------------
;
; Завдання:             Основи розробки і налагодження
; ВНЗ:                  КПІ ім. Ігоря Сікорського
; Факультет:    ФІОТ
; Курс:                 1
; Група:        __ __ __
;-------------------------------------------------------------------------
; Автор:        __ __ __
; Дата:         __/__/__
;-------------------------------
                                        TITLE     STRMENU (EXE)
.MODEL SMALL
.STACK 64
.DATA
;-------------------------------------------------------------------
TOPROW     EQU    08          ; Верхній рядок меню
BOTROW     EQU    15          ; Нижній рядок меню
LEFCOL     EQU    26          ; Лівий стовпчик меню
ATTRIB     DB     ?           ; Атрибути екрана
ROW        DB     00          ; Рядок екрана
SHADOW     DB     19 DUP(0DBH);
MENU       DB     0C9H, 17 DUP(0CDH), 0BBH
DB     0BAH, ' Add records     ',0BAH
DB     0BAH, ' Delete records  ',0BAH
DB     0BAH, ' Enter orders    ',0BAH
DB     0BAH, ' Print report    ',0BAH
DB     0BAH, ' Update accounts ',0BAH
DB     0BAH, ' View records    ',0BAH
DB     0C8H, 17 DUP(0CDH), 0BCH
PROMPT     DB     'To select an item, use <Up/Down Arrow>'
DB     ' and press <Enter>.'
DB     13, 10, 'Press <Esc> to exit.'
.386  ;-----------------------------------------------------------------
.CODE
A10MAIN    PROC   FAR
MOV    AX,@data
MOV    DS,AX
MOV    ES,AX
CALL   Q10CLEAR         ; Очищення екрана
MOV    ROW,BOTROW+4
A20:
CALL   B10MENU          ; Виведення меню
MOV    ROW,TOPROW+1       ; Вибір верхнього пункту меню
                                        ;   у якості початкового значення
MOV    ATTRIB,16H       ; Переключення зображення в інв.
CALL   D10DISPLY        ; Відображення
CALL   C10INPUT         ; Вибір з меню
JMP    A20              ;
A10MAIN    ENDP
;-------------------------------------------------------------------
;    Виведення рамки, меню і запрошення…
;-------------------------------------------------------------------
B10MENU    PROC   NEAR
```

```
PUSHA                   ;
MOV    AX,1301H         ;
MOV    BX,0060H         ;
LEA    BP,SHADOW        ;
MOV    CX,19            ;
MOV    DH,TOPROW+1      ;
MOV    DL,LEFCOL+1      ;
B20:     INT    10H
;;;;;
INC    DH               ; Наступний рядок
CMP    DH,BOTROW+2      ;
JNE    B20              ;
MOV    ATTRIB,71H       ;
MOV    AX,1300H         ;
MOVZX  BX,ATTRIB        ;
LEA    BP,MENU          ;
MOV    CX,19
MOV    DH,TOPROW        ; Рядок
MOV    DL,LEFCOL        ; Стовпчик
B30:
INT    10H
ADD    BP,19            ;
INC    DH               ;
CMP    DH,BOTROW+1      ;
JNE    B30              ;
MOV    AX,1301H         ;
MOVZX  BX,ATTRIB        ;
LEA    BP,PROMPT        ;
MOV    CX,79            ;
MOV    DH,BOTROW+4      ;
MOV    DL,00            ;
INT    10H
POPA                    ;,
RET
B10MENU    ENDP
;-------------------------------------------------------------------------
;  Натискання клавіш, управління через клавиші й ENTER
;  для вибору пункту меню і клавіші ESC для виходу
;-------------------------------------------------------------------------
C10INPUT   PROC    NEAR
PUSHA                   ;
C20:       MOV     AH,10H          ; Запитати один символ з кл.
INT     16H             ;
CMP     AH,50H          ; Стрілка донизу
JE      C30
CMP     AH,48H          ; Стрілка догори?
JE      C40
CMP     AL,0DH          ; Натистнено  ENTER?
JE      C90
CMP     AL,1BH          ; Натиснено  ESCAPE?
JE      C80             ;   Вихід
JMP     C20             ; Жодну ненатиснено, повторення
C30:
MOV     ATTRIB,71H      ; Колір символів
CALL    D10DISPLY       ;
INC     ROW             ;
CMP     ROW,BOTROW-1    ;
```

```
JBE     C50             ;
MOV     ROW,TOPROW+1    ;
JMP     C50
C40:
MOV     ATTRIB,71H      ; Колір символів і екрана
CALL    D10DISPLY       ;
;
DEC     ROW
CMP     ROW,TOPROW+1    ;
JAE     C50             ;
MOV     ROW,BOTROW-1    ;
C50:
MOV     ATTRIB,17H      ; Колір символів
CALL    D10DISPLY       ;
JMP     C20
C80:
MOV     AX,4C00H
INT     21H
C90:
POPA
RET
C10INPUT    ENDP
;-------------------------------------------------------------------------
;  Забарвлення виділеного рядка
;-------------------------------------------------------------------------
D10DISPLY  PROC    NEAR
PUSHA
MOVZX   AX,ROW
SUB     AX,TOPROW
IMUL    AX,19
LEA     SI,MENU+1
ADD     SI,AX
MOV     AX,1300H
MOVZX   BX,ATTRIB
MOV     BP,SI
MOV     CX,17
MOV     DH,ROW
MOV     DL,LEFCOL+1
INT     10H
POPA
RET
D10DISPLY  ENDP
;-------------------------------------------------------------------------
;    Очищення екрана
;-------------------------------------------------------------------------
Q10CLEAR   PROC    NEAR
PUSHA
MOV     AX,0600H
MOV     BH,61H
MOV     CX,00
MOV     DX,184FH
INT     10H
POPA
RET
Q10CLEAR   ENDP
END     A10MAIN
```

Якщо кнопка не відпущена, рух курсору здійснюється циклічно. Під час натискання кнопки клавіатури «стрілка донизу», здійснюються такі самі функції, тільки циклічно зверху донизу. Під час натискання «Enter» здійснюється виділення відповідного елемента меню. Під час натискання «Esc» здійснюється вихід з програми. Після виходу з програми кольори екрана залишаються такими самими.

Основний цикл програми представлений підпрограмою *A10MAIN*. Як можна побачити з коду, в основному циклі *A10MAIN* викликаються по черзі чотири підпрограми. Перша реалізує очищення екрана.

```
.....
CALL Q10CLEAR       ; очищення екрана
```

Далі зверніть увагу на мітку, що призначена для основного циклу програми *A20*. Вона визначає межі основного циклу програми, в якому реалізуються три основні підпрограми.

```
A20:          …
    CALL B10MENU        ; виклик меню
   …
    CALL D10DISPLY      ; відображення елементів
  …
    CALL C10INPUT       ; управління меню
    JMP          A20    ; основний цикл програми
```

Після реалізації всіх підпрограм управління повертається на мітку A20.

Вихід з програми здійснюється через C10INPUT, про що буде докладніше пояснено далі.

Підпрограми B10MEN, D10DISPLY призначені для відображення меню на екрані й їх можна вивчити з використанням файлу STRMENU.asm.

Важливим елементом програми є підпрограма управління програмою C10INPUT, що викликається у головному циклі. Розглянемо її більш докладно. У ній застосовано переривання DOS INT 16H, що реалізує введення з клавіатури сигналу (символа).

Отже, у реальному режимі роботи, користуючись перериваннями DOS, можна створювати псевдографічний інтерфейс користувача.

## *Перелік запитань для підготовки до лабораторної роботи*

1. Опишіть типи, призначення регістрів і організацію пам’яті мікропроцесорів архітектури AMD64 (Intel® 64).

2. Перерахуйте регістри загального призначення мікропроцесорів архітектури AMD64 (Intel® 64). Для чого призначені регістри, які їх особливості.

3. Перерахуйте сегментні регістри архітектури AMD64 (Intel® 64). Що зберігається у цих регістрах в архітектурі AMD64 (Intel® 64).

4. Перерахуйте системні регістри мікропроцесорів архітектури AMD64 (Intel® 64). Для чого призначені регістри, які їх особливості?

5. Як здійснюється адресація у мікропроцесорах архітектури AMD64 (Intel® 64) у реальному і захищеному режимах, навіщо розроблений цей механізм?

6. Опишіть типи, призначення регістрів і організацію пам’яті мікропроцесорів архітектури ARM64.

# ОФОРМЛЕННЯ ЗВІТУ ТА ПОРЯДОК ЙОГО ПОДАННЯ

Звіт можна подавати в електронній формі або у роздрукованому вигляді, бажано без затримок. Затримка не повинна перевищувати одну лабораторну роботу. У випадку більшої затримки оцінка знижується на 1 бал. Під час виконання роботи на Асемблері необхідною умовою захисту лабораторної роботи є:

– наявність файлу з *вихідним кодом*;

– у вихідному коді коментарі з прізвищем, номером групи, номером лабораторної роботи;

– наявність файлу з лістингом програми;

– здатність студента зробити транслювання файлу вихідного коду;

– здатність студента зробити компонування програми;

– здатність студента здійснити покроковий запуск програми під час налаштування, при цьому студент повинен пояснити змінні в пам’яті програми під час кожного кроку; пояснити призначення кожної інструкції. У разі необхідності до програми додаються фотографії екранів і відповідні обчислення, що обумовлено у кожній лабораторної роботі.

Код повинен містити необхідні коментарі, його дозволено використовувати під час захисту роботи. У випадку помилок під час асемблювання програми або під час її запуску робота до захисту не допускається.

У звіті має бути висновок, що містить інформацію про укладення, що виникли під час розробки і налагодження програми, відповідність результатів теоретичним положенням.

Титульний аркуш для звіту подано далі.

Міністерство освіти і науки України
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра автоматики та управління в технічних системах

# ЛАБОРАТОРНА РОБОТА № 1

# ТЕХНОЛОГІЯ РОЗРОБЛЕННЯ ПРОГРАМ У REAL MODE

## З ДИСЦИПЛІНИ «Архітектура комп'ютера»

**Виконав:**
студент групи ІТ – 91мн
_____
«__»_____

**Перевірив:**
доцент
_____
«__»_____

Київ – 2021

# СПИСОК ЛІТЕРАТУРИ

*Основна література*

1. Advanced Micro Devices, Inc. AMD64 Architecture Programmer's Manual Volume 1: Application Programming. Publication No. 24592. Revision Date 3.22. December 2017[Electron resource]. – Access link: https://archive.org/details/ /advancedmicrodevices_24592_3.22/mode/2up.

2. Intel® Corporation. Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4. Submitted: May 01, 2018 Last updated: May 27, 2020 [Electron resource]. – Access link: https://software.intel.com/content/www/us/en/develop/download/ /intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4.html.

3. Таненбаум Э. Архитектура компьютера / Э. Таненбаум, Т. Остин. – Изд. 6-е. – СПб.: Питер, 2013. – 816 с.: ил.

4. Рудаков П. И. Программируем на языке ассемблера IBM PC / П. И. Рудаков, К. Г. Финогенов. – Изд. 2-е. – ОбНШиск: Изд-во «Питер», 1997. – 584 с.

5. 8th and 9th Generation Intel® Core Prosessor families and Intel® Xeon E Prosessor familie. Datasheet, vol. 2 of 2. May 2020. Rev® 003.

*Допоміжна література*

1. Програмування 3. Системне програмування [Електронний ресурс]: метод. рекомендації до викон.лаб. робіт для студ. кафедри Автоматики та управління у техн. системах заочної форми навчання / НТУУ «КПІ» ; Уклад. П. Ю. Катін. – 2-ге вид., випр. і доп. – Електронні текстові дані (1 файл: 432 Кбайт). – Київ : НТУУ «КПІ», 2015. – 73 с. – Назва з екрана.

2. Тарарака В. Д. Т19 Архітектура комп'ютерних систем : навч. посіб. / В. Д. Тарарака. – Житомир : ЖДТУ, 2018. – 383 с.

3. Сучасні напрямки комп'ютерної та мікропроцесорної техніки. Розділ 1. Основні тенденції розвитку комп'ютерної і мікропроцесорної техніки. Розділ 2 Характеристики ARM і Cortex процесорів: конспект лекцій [Електронний ресурс]: для студ. спец. 171 Електроніка, спеціалізації

«Електронні компоненти та системи» / Уклад. : Т. О. Терещенко, Ю. С. Ямненко; КПІ ім. Ігоря Сікорського. – Електронні текстові данні (1 файл: 5,248 Мбайт). – Київ: КПІ ім. Ігоря Сікорського, 2020. – 68 с.

4. INTEL 80386 PROGRAMMER'S REFERENCE MANUAL/ INTEL CORPORATION. 1986. – 421 с. [Electron resource]. – Access link: https://css.csail.mit.edu/6.858/2014/readings/i386.pdf.

5. Абель П. Язык программирования для IBMPC [пер. с англ.] / П. Абель. – Киев : Век+., М. : ЭНТРОП, Киев : НТИ, 2003. – 736 с.

## Додаток 1
## Опис механізмів адресації

```
%TITLE "Приклад адресацій"
ideal
model   small
stack   256

dataseg

; Опис структури

struc  employee ;struc – директива (псевдокоманда) TASM для визначення
;структури даних; employee (службовець) – назва (ім'я) структури
dept    db  1 ;поле (елемент) структури – dept (департамент), завдовжки в 1 байт
;і початково встановленим значенням = 1
div_    db  2 ;поле (елемент) структури – div (відділ), завдовжки в 1 байт
;і початково встановленим значенням = 2
vac     db  3 ;поле (елемент) структури – vac (вакансія), завдовжки в 1 байт
;і початково встановленим значенням = 3
array   dw  7 dup (0eeeeh);поле структури – array (масив), яке визначає
;масив з 7 змінних, завдовжки в слово
;і початковим значенням = 0EEEEh
rate    dw  0ffffh ;поле (елемент) структури - rate(ставка), завдовжки в 2 байта
;і початково встановленим значенням = 0FFFFh
age     db  21     ;поле (елемент) структури - age(вік), завдовжки в 1 байт
;і початково встановленим значенням = 21
status  db  6      ;поле (елемент) структури – status (статус), завдовжки в 1 байт
;і початково встановленим значенням = 6
ends   employee    ;ends – директива (псевдокоманда) TASM для визначення кінця
;структури даних;employee (службовець) – назва (ім'я) структури

array2Da db  0,1,2,3    ;двовимірний масив array2Da, що складається з трьох рядків
db  4,5,6,7    ;і чотирьох стовбчиків
db  8,9,10,11

Maxim  employee <>;оголошення структури employee (Maxim - мітка)
;<> - вказує TASM на те, що значення полей (елементів)
;структури рівні тим, котрі були початково встановлені під час
```

```
;опису структури
Alex  employee <2,4,5,,5555h,22,5>;оголошення структури employee
;(Alex - мітка)
;<2,4,5,,5555h,22,5> - вказує TASM на те, що
;значення полів структури рівні:
;dept = 2
;div = 4
;vac = 5
;array = значенню, яке було встановлено під час опису структури
;rate =  5555h
;age =  22
;status = 5

exCode   db  0         ;ім'я змінної exCode резервується для запису коду
;помилки, якщо виникне помилка і виконання програми
;буде перервано. В цьому разі код помилки записується
;в комірку exCode і виконується команда JMP Exit.
dataByte db  99        ;ініціалізована змінна, завдовжки в один байт
;і значенням 99D.
dataWord dw  0FACEh    ;ініціалізована змінна, завдовжки у два байти
;(слово) і значенням FACEh.
array1   db  'a','b','c','d','e','f';одновимірний масив з шести символьних
;констант a, b, c, d, e, f.
array2Db db  1,0,0,0   ;двовимірний масив array2Db, що складається з трьох рядків
db  1,1,0,0    ;і чотирьох стовбчиків
db  1,1,1,1

copyright db  'Copyright (c) 1998-2000 Max  Natali IA-82',0 ; рядкова
;змінна copyright (ASCIIZ - рядок)
cpy_len = $ - copyright  ;макровизначення (константа) cpy_len, що визначає
;довжину рядкової змінної copyright, де
;$ - адреса наступної команди, а
;copyright - адреса рядкової змінної
string    db  'left ' ;рядок (масив) з п'яти символьних констант: l,e,f,t, .
str_len = $ - string  ;макровизначення (константа) str_len, що визначає
;довжину рядкової змінної string, де
;$ - адреса наступної комірки dataseg, а
;string - адреса рядка (масива)

codeseg

codeByte DB 11     ;ініціалізована змінна codeByte, завдовжки в 1 байт
;і початково встановленим значенням = 11
codeWord DW 0DECFh ;ініціалізована змінна codeWord, завдовжки в слово
;і початково встановленим значенням = DECFh

Start:             ;точка входу в програму. Визначається в кінці програми
;рядком END START.

mov     ax, @data  ;ax<-@data
; @data- 16-розрядна                    ;\
; константа(зарезервоване слово) - логічна адреса ; \
; сегмента DS                           ;  > ініціалізація ds і es
mov     ds, ax     ;ds<-ax             ; /
mov     es, ax     ;es<-ax             ;/
```

```
; Безпосередня адресація

mov ah, 1           ;Пересилання байта=1 в ah (ah<-1)
mov bx, 0FFh        ;Пересилання слова=0FFh в bx (bx<-0FFh)

add al, 85h         ; а)  al<-al+85h
add ax, 7a85h       ; б)  ax<-ax+7a85h

add dl, 0b6h        ;в)  dl<-dl+0b6h
add dx, 2fb6h       ; г)  dx<-dx+2fb6h

add dx, 0ffb6h      ;рис. 20 д)  dx<-dx+0ffb6h
;Якщо значення усіх розрядів СБ БО збігаються зі старшим бітом
;МБ БО, то компілятор оптимізує машиний код, включаючи в
;нього тільки МБ БО, який під час виконання команди розширюється
;до 2-х байт значенням старшого знакового біта МБ БО (див. лістинг)

mov bx, offset dataWord ;Завантаження адреси (зміщення в сегменті DS) змінної
;dataWord в bx
mov si, offset dataByte ;Завантаження адреси (зміщення в сегменті DS) змінної
;dataByte в si

; Неявна адресація

xchg    ax, bx      ;Обмін між ax і bx (ax <-> bx)
;Неявно адресується регістр ax

; Пряма адресація

mov al, [dataByte]    ; а)  al<-M(DSx16+offset dataByte)
mov ax, [dataWord]    ; б)  al<-M(DSx16+offset dataWord)
;ah<-M(DSx16+offset dataWord+1)
mov bl, [dataByte]          ;bl<-M(DSx16+offset dataByte)

mov [dataWord], dx    ; в) ; M(DSx16+offset dataWord)<-dl
; M(DSx16+offset dataWord+1)<-dh
inc [dataByte]    ;M(DSx16+offset dataByte)<-M(DSx16+offset dataByte)+1
ror [word dataWord],1  ;циклічний зсув вправо на один розряд
;двобайтової змінної dataWord
jmp far very_far_label ; г) команда дальнього безпосереднього переходу
;на мітку very_far_label, яка знаходиться в тому ж
;самому сегменті та є ближньою, тому
;під час асемблерування (див. лістинг)
;виникає оптимізація цієї команди, тобто
;замість КОП 0eah (код дальнього переходу),
;tasm підставить 0ebh - машинний
;код команди ближнього безпосереднього переходу, а
;04h=disp8 - різниця між адресою мітки (0041h)
;й адресою наступної за jmp, команди NOP (003Dh)

very_far_proc:
retf                ;команда повернення з дальньої підпрограми

very_far_label:
call far very_far_proc ;команда виклику дальньої підпрограми з міткою
;very_far_proc.
```

```
;компілятор оптимізує цю команду і замінює
;її двома командами: кодом команди ближньої
;підпрограми (E8h),FFFBh=disp - різниця між
;адресою мітки (0040h) і адресою наступної команди
;(0045h), що дорівнює (-5)
;перед кодом E8h компілятор поміщає код команди
;PUSH CS (0Eh), щоб зберегти значення регістра CS
;у стеку.(див. лістинг програми)

; Регістрова адресація

mov cx, ax          ;пересилання слова з ax в cx
mov dx, ds          ;пересилання слова з ds в dx

inc bx              ;а) bx<-bx+1
mov dh, ch          ; б) dh<-ch
mov ch, dh          ; в) ch<-ds

mov si, bp          ;г) si<-bp
mov bp, si          ; д) bp<-si

mov si, ds          ;е) si<-ds
mov ds, si          ; ж) ds<-si

; Непряма адресація

mov ax, [di]          ;al<-M(DSx16+di)
;ah<-M(DSx16+di+1)
or  al, [bp]          ;al<-al v M(SSx16+bp)
mov cl, [bx]          ; б) cl<-M(DSx16+bx)
mov [si], cl          ; в) M(SSx16+si)<-cl
mov ax, offset jmp_label  ;ax<-offset jmp_label
jmp ax              ;г) команда ближнього безпосереднього переходу
;на адресу, що зберігається в регістрі
;ax (offset jmp_label)

xchg    ax, ax    ;ax<->ax      \ однаковий КОП (див. лістинг)
nop               ;порожня операція  /

jmp_label:

mov ax, [bx]        ;пересилання слова в ax з сегмента даних зі зміщенням,
;що знаходиться в bx

; Базова адресація

mov    ah, [bp + 77h]      ;б) ah<-M(SSx16+BP+0077h), де
;BP=var - база, disp8=77h=const - індекс
mov    ah, [bx + 0fefch]   ; в) ah<-M(DSx16+BX+0FEFCh), де
;BX=var - база, disp16=0FEFCh=const - індекс

mov    bx, offset Maxim    ;г) bx<-адреса структури employee
;з міткою Maxim
mov    ax, [bx + 7*2 + 3]  ;ax<-[Maxim.rate]
;пересилання в ax поля (елемента) rate
;структури employee з міткою Maxim
```

```
mov     bx, offset Alex       ; г) bx<-адреса структури employee
;з міткою Alex
mov     ax, [bx + 7*2 + 3]    ;ax<-[Alex.rate]
;пересилання в ax поля (елемента) rate
;структури employee з міткою Alex

mov     [byte bx + 2],0DDh    ;[Alex.vac] <- 0DDh
;Завантаження в поле (елемент) vac
;структури employee з міткою Alex
;константи байта 0DDh

; індексна адресація

mov     si, 2                 ; в) si <- 2
mov     di, 5                 ;di <- 5
mov     [array1+si],16        ;array1(2) <- 16
;M(DSx16+offset array1+si)<-16, де
;offset array1=disp16=003A=const - база
;масива, si=var - індекс, що адресує елемент
;масива (в цьому прикладі адресується другий
;елемент (si=2) починаючи з нульового)
mov     [array1+di],0EDh      ;array1(5) <- 0EDh
;M(DSx16+offset array1+di)<-16, де адресується
;п'ятий елемент масиву array1 (di=5)
mov     ah,[array1+di]        ;ah <- array1(5), пересилання п’ятого елемента
;масива array1 в регістр ah

; Базово-індексна адресація

mov     bx, 1                 ; пересилання в регістр bx номера
;стовбчика (№ стовбчика=1) двовимірного масиву
;array2Da (номери стовбчиків змінюються
;від 0 до 3) (BX - база)
mov     di, 2*4               ;пересилання в регістр di добутку
;номера рядка (№ рядка=2) на число
;елементів у рядку (кількість елементів=4)
;двовимірного масиву 2Da (DI - індекс)
mov     al, [di+bx]           ;al<-array2Da[2][1] (пересилання в регістр
;al значення елемента масиву array2Da,
;що лежить на перехресті другого рядка
;й першого стовбчика); масив array2Da
;Оголошений на початку сегмента даних і
;має нульове зміщення
dec     bx                    ;bx<-bx-1
mov     al, [di+bx]           ;al<-array2Da[2][0] (пересилання в регістр
;al значення елемента двовимірного масиву
;array2Da, що лежить на перехресті другого
;рядка й нульового стовбчика)

; Базово-індексна адресація зі зміщенням

mov     bx, offset Maxim    ; bx<-offset Maxim (завантаження в bx
;адреси структури employee з міткою Maxim)
mov     si, 3*2             ;завантаження в si індекса, що дорівнює 3*2
mov     dx, [bx+3+si]       ;пересилання в регістр dx третього елемента
;(si=3*2 т.к. елемент масиву - слово)
```

```
;масиву array структури employee з
;міткою Maxim
mov     bx, offset Alex     ;bx<-offset Alex (завантаження в bx адреси
;структури employee з міткою Alex
mov     si, 5               ;завантаження в si індекса, що дорівнює 5
mov     bl, [bx+3+si]       ;пересилання в регістр bl старшого байта
;другого елемента
;(si=5) масиву array структури employee
;з міткою Alex
mov     bx, 3               ;пересилання в регістр bx номера стовбчика
;(№ стовбчика=3) двовимірного масиву array2Db
mov     di, 2*4             ;пересилання в регістр di добутку номера
;рядка (№ рядка=2) на кількість елементів у
;рядку (кількість елементів = 4) двовимірного
;масиву array2Db
mov     ch, [offset array2Db+di+bx] ;ch<-array2Db[2][3] пересилання в регістр
;ch значення елемента двовимірного масиву
;array2Db, що знаходиться на перехресті другого рядка
;й третього стовбчика

; Стекова адресація

push    ax                  ;рис.  M(SSx16+SPпоч-1)<-ah
;       M(SSx16+SPпоч-2)<-al
;       SP<-SPпоч-2
push    bx                  ;       M(SSx16+SPпоч-1)<-bh
;       M(SSx16+SPпоч-2)<-bl
;       SP<-SPпоч-2

pop     ax                  ;       al<-M(SSx16+SPпоч)
;       ah<-M(SSx16+SPпоч+1)
;       SP<-SPпоч+2
pop     bx                  ;       bl<-M(SSx16+SPпоч)
;       bh<-M(SSx16+SPпоч+1)
;       SP<-SPпоч+2



back_jump:
jmp     short forvard_jump  ; безумовний короткий перехід на мітку
;forvard_jump, компілятор (див. лістинг)
;формує два байти: 1-й - КОП команди (EBh),
;2-й - disp8=адреса мітки-адреса наступної
;команди=00C2h-00C0h=0002h (2D)
jmp     back_jump           ;безумовний перехід на мітку back_jump
;компілятор (див. лістинг) формує
;два байти: 1-й - КОП команди (EBh)
;2-й - disp8=адреса мітки-адреса наступної
;команди=00BEh-00C2h=FCh (-4D)
forvard_jump:

; адресація строк даних

cld                         ; DF<-0 - обнуління прапорця напрямку
;оброблення рядків
mov cx, str_len             ;завантаження в регістр cx константи str_len,
```

```
;що визначає довжину рядка string
;(str_len=5D див. лістинг)
mov di, offset copyright+4   ;завантаження в регістр di адреси 4-го елемента
;ASCII рядка "copyright"
mov si, offset string        ;завантаження в регістр si адреси початку
;нульового елемента ASCII рядка string
rep movsb                    ;пересилання елементів рядка string на місце
;п'яти елементів рядка copyright, починаючи з
;4-го елемента (замінює слово copyright на
;слово copyleft)

; Пересилання рядка copyright в відеобуфер

mov     ax, 0b800h           ;Сегментна адреса відеобуфера пересилається в
;регістр ax
mov     es, ax               ;es<-ax (ax=сегментній адресі відеобуфера)
mov     ah, 1eh              ;завантаження в регістр ah атрибута елементів
;рядка (жовтий на синьому)
mov     si, offset copyright ;завантаження в регістр si адреси початку рядка
;copyright
mov     di, 80*2*1           ;Зміщення в відеобуфері в байтах (80 символів
;в рядку по 2 байта: МБ - код символа,
;СБ - код його атрибута). В цьому випадку - це
;1-ий рядок, 0-ий стовбчик
mov     cx, cpy_len          ;завантаження в регістр cx константи cpy_len,
;що визначає довжину рядка copyright
;(cpy_len=2Eh=46D, див. лістинг)
cld                          ;DF<-0 (обнуління прапорця напрямку обробки
;рядків)
;використані нижче команди роботи з рядками
;будуть інкрементувати регістри si й di
@@stor:
lodsb                        ;al<-M(DSx16+si)
stosw                        ;M(ESx16+di)<-al
;M(ESx16+di+1)<-ah
loop    @@stor               ;cx<-cx-1, якщо cx<>0, то перехід на мітку
;@@stor, компілятор (див. лістинг) формує
;два байта: 1-й - КОП команди (E2h)
;2-й - disp8=адреса мітки-адреса наступної
;команди=00DFh-00E3h=FCh (-4D)

; Адресація портів введення/виведення

in      al, 20               ; введення байта в акумулятор al з порту
;з адресою 20D=14h
;Команди, що використовують адресацію портів введення/виведення,
;використовуються в програмі управління світлодіодами клавіатури (inout.asm)

;Переназначення сегментних регістрів

mov     bl, [cs:codeByte]   ;bl<-M(CSx16+offset codeByte)\ сегмент DS
;
mov     [cs:codeByte], bh   ;M(CSx16+offset codeByte)<-bh/ на сегмент CS

Exit:                        ;Стандартний вихід в DOS з виконуваної
;програми
```

```
mov     ah,04Ch         ;ah <- 04Ch
;4Ch - функція DOS (завершення підпроцеса
;з поверненням управління програмі command.com)
mov     al,[exCode]     ;al <- M(DS*16+offset exCode)
;запис в al кода помилки
int     21h             ;переривання DOS, що викликає функцію DOS,
;яка визначається кодом в регістрі ah
END Start                       ;кінець програми Start
```

# Додаткок 2
# Таблиці випадкових чисел

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 10097 | 74296 | 15953 | 27659 | 03529 | 65813 | 09732 | 27732 |
| 37542 | 09303 | 88676 | 76833 | 76850 | 86799 | 88579 | 98083 |
| 08422 | 70715 | 98951 | 66065 | 82406 | 73053 | 25624 | 58401 |
| 99019 | 36147 | 09117 | 31060 | 65692 | 28468 | 88435 | 64969 |
| 12807 | 34673 | 10402 | 85269 | 47048 | 60935 | 98520 | 22109 |
| 32533 | 24805 | 34764 | 63573 | 64778 | 39885 | 11805 | 50725 |
| 04805 | 23209 | 74397 | 73796 | 36697 | 07439 | 83452 | 13746 |
| 68953 | 38311 | 16877 | 74717 | 35303 | 85247 | 88685 | 36766 |
| 02529 | 64032 | 39292 | 10805 | 68665 | 28709 | 99594 | 91826 |
| 99970 | 54876 | 00822 | 77602 | 90553 | 20344 | 17767 | 40558 |
| 76520 | 24037 | 35080 | 32135 | 35808 | 11199 | 05431 | 68248 |
| 64894 | 02560 | 04436 | 45753 | 36170 | 23403 | 99634 | 70078 |
| 19645 | 31165 | 12171 | 34072 | 42614 | 18623 | 40200 | 67951 |
| 09376 | 36653 | 74945 | 45571 | 74818 | 83491 | 67348 | 08928 |
| 80157 | 80959 | 91665 | 02051 | 57548 | 35273 | 14905 | 60970 |
| 13586 | 20636 | 33606 | 05325 | 34282 | 29170 | 39808 | 29405 |
| 18475 | 11062 | 14225 | 73039 | 53763 | 85157 | 74697 | 07207 |
| 90364 | 34113 | 50950 | 21115 | 02655 | 11100 | 52563 | 20048 |
| 93785 | 65481 | 08015 | 45521 | 56418 | 16505 | 94750 | 82341 |
| 93433 | 80124 | 77214 | 76621 | 14598 | 27686 | 53140 | 06413 |
| 24201 | 74350 | 29148 | 96297 | 91499 | 36858 | 57600 | 25815 |
| 40610 | 69916 | 68514 | 57186 | 80336 | 47954 | 96644 | 31790 |
| 76493 | 09893 | 58047 | 78253 | 44104 | 02040 | 43651 | 61196 |
| 61368 | 17674 | 45318 | 64237 | 12550 | 34484 | 89923 | 15474 |
| 50500 | 35635 | 43236 | 13990 | 63606 | 46162 | 33340 | 94557 |
| 52775 | 99817 | 36936 | 78822 | 14523 | 70297 | 40881 | 42481 |
| 68711 | 26803 | 46427 | 40218 | 94598 | 32979 | 89439 | 23523 |
| 29609 | 20505 | 76974 | 14385 | 81949 | 12860 | 77082 | 90446 |
| 23478 | 17468 | 22374 | 96286 | 73742 | 40219 | 37089 | 45266 |
| 73998 | 17727 | 00210 | 94400 | 49329 | 83554 | 42050 | 28573 |
| 67851 | | | | | | | |
| 77817 | | | | | | | |

Навчальне видання

**Катін** Павло Юрійович

# Архітектура комп’ютера

## Лабораторний практикум

Електронне мережеве видання

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Свідоцтво про державну реєстрацію: серія ДК № 5354 від 25.05.2017 р.
просп. Перемоги, 37,
м. Київ, 03056

Гарнітура Times. Поз. 20-2-005.

Видавництво «Політехніка» КПІ ім. Ігоря Сікорського
вул. Політехнічна, 14, корп. 15
м. Київ, 03056
тел. (044) 204-81-78