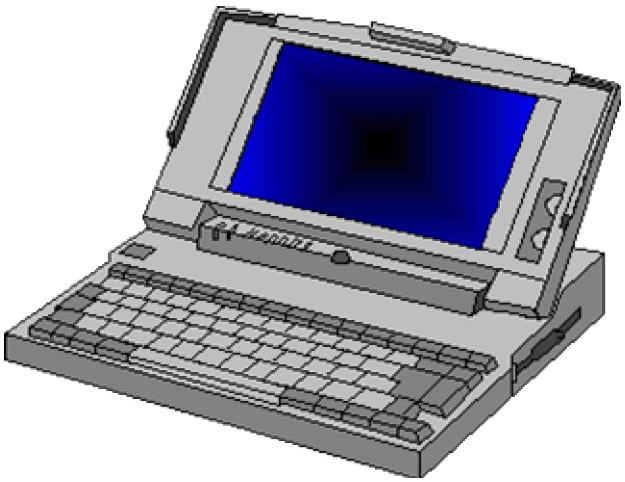
ДЕРЖАВНИЙ ВИЩИЙ НАВЧАЛЬНИЙ ЗАКЛАД
«ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ»
МІНІСТЕРСТВА ОСВІТИ І НАУКИ УКРАЇНИ

С.В. Чопоров

ОРГАНІЗАЦІЯ І ФУНКЦІОНУВАННЯ ЕОМ

Навчально-методичний посібник до лабораторних занять
для студентів освітньо-кваліфікаційного рівня «бакалавр»
напряму підготовки «Програмна інженерія»

Затверджено
Вченою радою ЗНУ
Протокол № від р.



Запоріжжя
2014

УДК 004 (075.8)
ББК З973.2я73
Ч754

Чопоров С.В. Організація і функціонування ЕОМ: навчально-методичний посібник до лабораторних занять для студентів освітньо-кваліфікаційного рівня «бакалавр» напряму підготовки «Програмна інженерія» / С.В. Чопоров. – Запоріжжя: ЗНУ, 2014. – 72 с.

Навчально-методичний посібник до виконання лабораторних робіт з дисципліни «Організація і функціонування ЕОМ» пропонується студентам, що навчаються за програмою освітньо-кваліфікаційного рівня «бакалавр» за напрямом підготовки «Програмна інженерія» та рекомендується для закріплення навчального матеріалу з дисципліни циклу професійної та практичної підготовки.

Посібник включає теоретичний матеріал з кожної теми дисципліни та зміст основних лабораторних робіт, що пропонуються при вивченні даного курсу, а також контрольні питання. Увага студентів акцентується на розширенні теоретичних знань та набутті практичних навичок системного програмування мовою асемблера.

Навчально-методичний посібник призначений для студентів, викладачів, а також усіх, хто цікавиться питаннями зазначеної тематики.

Рецензент *С.Ю. Борю*

Відповідальний за випуск *С.І. Гоменюк*

Зміст

Вступ

Організація і функціонування ЕОМ – це область комп'ютерної науки і технології, яка займається вивченням особливостей будови всіх рівнів функціонування комп'ютерної техніки, особливостей взаємодії функціональних елементів, організації взаємодії програмного та апаратного забезпечення.

Мета курсу: набуття теоретичних знань, необхідних для використання основних підходів організації і функціонування ЕОМ, та формування практичних навичок з їх реалізації засобами системного програмування мовою асемблера у студентів напряму підготовки «Програмна інженерія», які можуть бути використані при подальшому навчанні, професійній, виробничій та науковій діяльності випускника вузу. Дисципліна «Організація і функціонування ЕОМ» базується на знаннях дисциплін «Методи та засоби комп'ютерних інформаційних технологій», «Основи програмування та інформаційна культура студентів», «Архітектура комп'ютера» та інших, що викладаються за програмою підготовки бакалавра галузі знань 0501 – «Інформатика та обчислювальна техніка».

Навчально-методичний посібник містить 12 лабораторних робіт, у яких потрібно розробити програмні засоби мовою асемблера, встановити зміни стану регістрів та пам'яті комп'ютеру шляхом стеження на базі зневаджувача.

Завдання навчальної дисципліни: оволодіти основними принципами та методами організації і функціонування сучасних комп'ютерів. Дати необхідну практичну підготовку та знання для подальшого їх застосування при вивченні спеціальних дисциплін та професійній підготовці.

Загальні вказівки

1. Лабораторні роботи виконуються в години, зазначені в розкладі і для кожного студента присутність на занятті є обов'язковою. Студенту, що пропустив лабораторне заняття без поважних причин і не захистив лабораторну роботу на наступному занятті, виставляється незадовільна оцінка з відповідної лабораторної роботи.

2. У комп'ютерний клас дозволяється входити тільки після дзвінка на заняття й у присутності викладача.

3. Вхід у комп'ютерний клас дозволяється тільки за наявністю документа, завіреного відповідальним органом ЗНУ, що засвідчує особистість .

4. Лабораторну роботу припиняють виконувати за дзвінком.

5. Забороняється виходити з аудиторії без дозволу викладача.

Вимоги до виконання лабораторних робіт

1. Ознайомитися з загальними теоретичними відомостями.

2. За номером варіанта[1] обрати завдання.

3. Уважно прочитати завдання.

4. Кожну роботу виконувати відповідно до завдання.

5. Оформити друкований звіт з виконання лабораторної роботи, який повинен містити:

– титульний лист;

– тему роботи;

– схеми основних алгоритмів;

– послідовність виконання роботи;

– при необхідності записуються теоретичні відомості, результати втілення та аналіз отриманих результатів;

– висновки.

6. При необхідності захистити результати, відповісти на додаткові питання та пред'явити виконану роботу.

1 Номер варіанта визначає викладач.

Лабораторна робота № 1. Асемблювання та компонування програми

Теоретичні відомості

Асемблер NASM

NASM (Netwide Assembler) – вільний (LGPL та ліцензія BSD) асемблер для архітектури Intel x86. Використовується для написання 16-, 32- та 64-бітних програм.

NASM створений Саймоном Тетхемом разом з Юліаном Холлом та у тепершній час розвивається невеликою командою розробників на базі хостингу відкритих проектів на сайті SourceForge.net.

NASM може працювати на платформах, відмінних від x86, таких як SPARC та PowerPC, однак код він генерує лише для x86 та x86-64.

NASM досить успішно конкурує зі стандартним у Linux- та багатьох інших UNIX-системах асемблером gas. Вважається, що якість документації у NASM вище, ніж у gas. Крім того, асемблер gas за замовченням використовує AT&T-синтаксис, орієнтований на процесори не від Intel, у той час як NASM використовує варіант традиційного для x86-асемблерів Intel-синтаксису; Intel-синтаксис використовується усіма популярними асемблерами для DOS/Windows, наприклад, MASM, TASM, fasm.

Компіляція програм

NASM компілює програми під різні операційні системи в межах x86-сумісних процесорів. Знаходячись в одній операційній системі, можна компілювати файл для виконання у іншій[2].

Компіляція програм в NASM складається з двох етапів. Перший – асемблювання, другий – компонування. На етапі асемблювання створюється об'єктний код. В ньому міститься машинний код програми та данні, у відповідності з вихідним кодом, але ідентифікатори (змінні, символи) поки не прив'язані до адрес пам'яті. На етапі компонування з одного або декількох об'єктних модулів створюється файл для виконання (програма). Операція компонування зв'язує ідентифікатори, визначені в основній програмі, з ідентифікаторами, визначеними в інших модулях, після чого всім ідентифікаторам додаються кінцеві адреси пам'яті або забезпечується їх динамічне призначення.

Для компонування об'єктних файлів у використовувані в Windows можна використати вільний (безкоштовно розповсюджуваний) компонувальник alink (для 64-бітних програм компонувальник GoLink), а в Linux – компонувальник ld, який є в будь-якій дистрибуції цієї операційної системи.

Для асемблювання файлу необхідно ввести наступну команду:

```
nasm -f format filename -o output
```

2 Можливість компіляції коду в одній операційній системі для іншої називається кроскомпіляцією.

де format – формат файлу з результатом, filename – ім'я файлу з вихідним кодом програми, output – ім'я файлу для збереження результату[3].

NASM підтримує множину форматів файлів з результатом, серед яких:

– bin – файл довільного формату, який визначається тільки вихідним кодом. Може бути використаним як для файлів даних, так і для модулів з кодами для виконання – наприклад, системних завантажувачів, образів ПЗП, модулів операційних систем, драйверів .SYS в MS-DOS або файлів виконання .COM.

– obj – об'єктний модуль у форматі OMF, сумісний з MASM і TASM.

– win32 та win64 – об'єктний модуль для 32- та 64-бітного коду, сумісний з Win32- та Win64-компіляторами Microsoft.

– aout – об'єктний модуль в форматі a.out, що використовувався в ранніх Linux-системах.

– aoutb – версія формату a.out для BSD-сумісних операційних систем.

– coff – об'єктний модуль у форматі COFF, сумісним з компонувальником з DJGPP.

– elf32 та elf64 – об'єктний модуль у форматах ELF32 та ELF64, які використовуються в Linux та Unix System V, включно Solaris x86, UnixWare та SCO Unix.

То б то, для створення об'єктного модулю для програми, збереженої в файлі hello.asm, на платформі Linux необхідно виконати команду

```
nasm -f elf32 hello.asm
```

для 32-бітної системи або, відповідно,

```
nasm -f elf64 hello.asm
```

для 64-бітної системи.

Аналогічно під керуванням Windows ці команди матимуть вигляд

```
nasm -f win32 hello.asm
```

для 32-бітної системи або, відповідно,

```
nasm -f win64 hello.asm
```

для 64-бітної системи.

Після виконання необхідної команди в поточній директорії з'явиться об'єктний файл, розширення якого відповідає цільовій платформі (наприклад, в системах на базі Linux з'явиться файл hello.o).

Для компонування програми за допомогою компонувальника ld необхідно виконати команду

```
ld -o output objectfile
```

3 Ключ -o та параметр output є необов'язковими. При їх відсутності NASM обирає ім'я файлом з результатом самостійно; тому що це залежить від формату об'єктного файлу. Якщо формат об'єктного файлу – Microsoft (obj та win32), він видалить розширення .asm (або інше, яке вам подобається використовувати – NASM працює з будь-яким розширенням файлу з текстом програми) з ім'я вихідного файлу та виконає заміну його на .obj. У об'єктних файлів Unix-формату (aout, coff, elf та as86) він буде заміняти розширення на .o. Для формату rdf він буде використовувати розширення .rdf, а у випадку формату bin він просто видалить розширення.

де `output` – ім'я файлу з результатом, `objectfile` – об'єктний файл. Наприклад, для компонування програми з об'єктним модулем `hello.o` необхідно виконати команду

```
ld -o hello hello.o
```

в результаті якої у поточній директорії з'явиться файл для виконання з назвою `hello`.

✍ *Завдання для лабораторної роботи*

Для наведеної нижче програми створити об'єктний модуль та провести компонування[4].

```
1   section .data
2       msg: db   "Hello, World!", 10
3       .len:equ  $ - msg
4   section .text
5       global main
6   main:
7       mov  EAX, 4     ; запис у
8       mov  EBX, 1     ; стандартний вивід
9       mov  ECX, msg
10      mov  EDX, msg.len
11      int  0x80       ; переривання операційної системи
12      mov  EAX, 1     ; вихід (завершення програми)
13      mov  EBX, 0     ; код виходу (статус)
14      int  0x80
```

Змінити код програми таким чином, що б на екрані друкувалися відомості про автора програми.

? *Контрольні запитання*

1. Дайте визначення, що таке машинна мова, знайдіть приклади декількох команд машинної мови для процесорів `Pentium`.
2. Що таке асемблер, навіщо він потрібен?
3. Поясніть особливості етапів асемблювання та компонування.
4. Дайте визначення термінів: інтерпретація, трансляція, компіляція.
5. Якому символу відповідає код 10?
6. Яким чином провести кроскомпіляцію для платформи `Win32` з платформи `Linux` використовуючи `NASM`?
7. Які можуть бути коди виходу програми (статуси завершення)?

4 Для виконання лабораторної роботи в середовищі операційної системи Windows рекомендується використовувати код простішого застосування, наведений за адресою: http://codewiki.wikispaces.com/winstdout.nasm (див. додаток А) або скористатися напрацюваннями проекту cygwin (http://www.cygwin.com/).

Лабораторна робота № 2. Використання зневаджувача для контролю стану регістрів

Теоретичні відомості

Зневаджувач[5] – це комп'ютерна програма, яка призначена для пошуку помилок в інших програмах, ядрах операційних систем, SQL-запитах та інших видах коду.

Зневаджувач дозволяє виконувати покрокове трасування, відстежувати, встановлювати або змінювати значення змінних в процесі виконання коду, встановлювати і знищувати контрольні точки або умови зупинки і т. д.

В контексті зневаджування програм, написаних на асемблері, важливою функцією зневаджувача є можливість перегляду стану регістрів процесора.

Одним з найбільш розповсюджених зневаджувачів є GNU Debugger – переносимий зневаджувач проекту GNU, який працює на багатьох UNIX-подібних системах і вміє виконувати зневадження багатьох мов програмування, включно C, C++, Free Pascal, FreeBASIC, Ada та Fortran.

Для початку зневадження програми необхідно виконати команду виду

```
gdb program_name
```

де program_name – назва бінарного файлу програми.

Наприклад, для зневадження програми з назвою файлу hello необхідно виконати команду

```
gdb hello
```

після чого з'явиться рядок очкування команд зневаджувача приблизно наступного вигляду[6]

```
GNU gdb (GDB) 7.4.1-debian
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and
redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/.../1/hello...(no debugging
symbols found)...done.
(gdb)
```

5 дебаггер, англ. debugger.

6 Кінцевий вигляд запрошення зневаджувача залежить від версії самого зневаджувача та операційної системи, в якій він запущений.

Для переведення зневаджувача в покроковий режим необхідно додати точку зупинки – оператор програми, після якого її виконання переривається до отримання наступних команд. Наприклад, в якості такої точки можна використовувати точку входу в програму (`main` в прикладі попередньої лабораторної роботи). Для цього необхідно виконати команду

```
break main
```

Для запуску виконання програми до найближчої точки зупинки необхідно виконати команду

```
run
```

Для отримання інформації про стан всіх регістрів необхідно виконати команду

```
info registers
```

В результаті виконання цієї команди на екран будуть виведені значення, які зберігаються в регістрах. Проте для перегляду стану одного окремого регістра необхідно виконати команду (у цьому випадку регістра `esp`)

```
print/x $esp
```

Для переходу зневаджувача до наступного оператора необхідно виконати команду

```
nexti
```

а для переходу до наступного рядка програми використовується команда `next`.

Необхідно зауважити, що для більш повного використання можливостей зневаджувача `gdb` необхідно програму компілювати з ключами включення додаткової інформації (`-g -Fstabs`). Наприклад, для компіляції програми під назвою `lab.asm` необхідно виконати команду

```
nasm -f elf -g -Fstabs lab.asm
```

✍ *Задання для лабораторної роботи*

На базі коду програми з лабораторної роботи № 1 сформувати звіт, який містить аналіз стану регістрів у зневаджувачі. Необхідно показати стан регістрів після виконання кожного оператору.

? *Контрольні запитання*

1. Що таке зневаджувач? Які його основні функції?
2. Наведіть назви та особливості відомих зневаджувачів.
3. Наведіть основні етапи процесу зневаджування.
4. Наведіть доступні команди зневаджувача `GNU Debugger`.
5. Які ключі команди `print` зневаджувача `gdb` можна використовувати? Наведіть приклади.

Лабораторна робота № 3. Регістри. Завантаження даних у регістри

Теоретичні відомості

Родина мікропроцесорів починаючи з моделі 80386 є повністю 32-розрядними, що означає те, що вони можуть обробляти 4 Гб оперативної пам'яті[7]. Оскільки шина даних також 32-розрядна, то процесор може зберігати і обробляти у своїх регістрах число "шириною" в 32 біти.

Для того, щоб програмувати на мові асемблера, необхідно знати імена регістрів та загальні принципи роботи команд завантаження даних.

Регістри загального призначення

Регістрами загального призначення є EAX (Акумулятор), EBX (База), ECX (Лічильник) та EDX (Дані).

Процесор 80386 зворотно сумісний з процесором 80286, регістри якого 16-розрядні. Регістри загального призначення мають однакову структуру, тому для розуміння достатньо розглянути структуру регістра EAX.

Регістр EAX може бути розділений на дві частини – 16-розрядний регістр AX (який також є в 80286) та старші 16 бітів, які не мають окремої назви. В свою чергу, регістр AX може бути розділеним (не тільки в 80386, проте і в 80286) на два 8-бітних регістри – AH та AL (рис. 1). Аналогічно, в регістрі EBX молодші 16 біт – регістр BX, який складається з регістрів BH та BL; ECX: CX = CH + CL; EDX: DX = DH +DL.

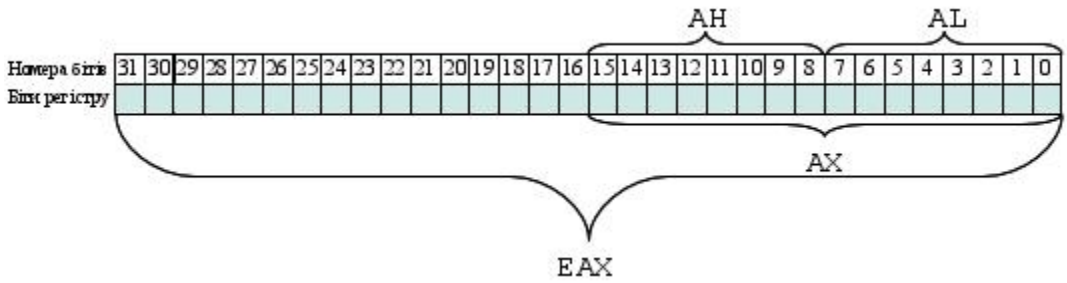


Рис. 1. Структура регістра EAX

Індексні регістри

До регістрів загального призначення іноді відносять також індексні регістри процесора 80386 – ESI, EDI та EBP (або SI, DI та BP для 16-розрядних дій).

Зазвичай ці регістри використовуються для адресації пам'яті: звертання до масивів, індексування, тощо. Звідси їх назви: індекс джерела (Source Index), індекс приймача (Destination Index), вказівник бази (Base Pointer). Але зберігати в них тільки адреси необов'язково: регістри ESI, EDI та EBP

7 32 біти можуть прийняти 2^{32} різні значення, тому можлива адресація 2^{32} байтів пам'яті, які утворюють 4 гігабайти.

можуть зберігати довільні дані. Ці регістри програмно доступні, то б то їх зміст може бути змінено програмістом. Інші регістри краще програмно не змінювати.

У регістрів ESI, EDI та EBP існують тільки в 16-розрядна та 32-розрядна версії.

Сегментні регістри

Цю групу регістрів можна віднести до регістрів стану. Регістри з цієї групи використовуються при обчисленні реальної адреси (адреси, яку буде передано на шину адреси). Процес обчислення реальної адреси залежить від режиму процесора (реальний або захищений). Сегментні регістри тільки 16-розрядні, такі ж, як в 80286.

Назви цих регістрів відповідають їх функціям: CS (Code Segment, сегмент коду) разом з EIP (IP) визначають адресу пам'яті, з якої потрібно прочитати наступну інструкцію; аналогічно регістр SS (Stack Segment, сегмент стека) в парі з ESP (SS:SP) вказують на вершину стека.

Сегментні регістри DS, ES, FS, та GS (Data, Extra, F та G сегменти) використовуються для адресування даних в пам'яті.

Команда завантаження даних

Загальний формат команд Асемблера наступний:

```
ім'я_команди [підказка] операнди
```

Для завантаження даних в регістр використовується команда mov. Команда MOV, не дивлячись на те, що її назва виходить зі слова «move» (переміщати), на справді не переміщує, а копіює значення з джерела в приймач. Загальний формат команди MOV наступний.

```
mov приймач, джерело
```

Розглянемо декілька прикладів застосування команди MOV.

```
mov AX, [number] ; в регістр AX копіюємо значення змінної number
mov [number], BX ; в змінну number завантажити значення з регістра BX
mov BX, CX    ; завантажити в регістр BX значення регістра CX
mov AL, 1     ; завантажити в регістр AL значення 1
mov DL, CL    ; завантажити в регістр DH значення регістра CL
mov ESI, EDI  ; копіювати значення регістра EDI в регістр ESI
mov word [number], 1; зберегти 16-бітне значення 1 в змінну number
```

Процесори родини x86 дозволяють використовувати в командах тільки один нерігстровий аргумент. Щоб скопіювати значення з однієї області пам'яті до іншої, необхідно використовувати проміжний регістр.

Перелік всіх можливих форматів команди MOV має вигляд, наведений нижче.

```
mov r/m8, reg8
mov r/m16, reg16
mov r/m32, reg32
mov reg8, r/m8
mov reg16, r/m16
mov reg32, r/m32
mov reg8, imm8
mov reg16, imm16
mov reg32, imm32
mov r/m8, imm8
mov r/m16, imm16
mov r/m32, imm32
```

де reg8 – будь-який 8-розрядний регістр загального призначення; reg16 – будь-який 16-розрядний регістр загального призначення; reg32 – будь-який 32-розрядний регістр загального призначення; r/m8 – 8-розрядний операнд може знаходиться в пам'яті або регістрі; r/m16 – 16-розрядний операнд може знаходиться в пам'яті або регістрі; r/m32 – 32-розрядний операнд може знаходиться в пам'яті або регістрі; imm8 – безпосередньо 8-розрядне значення; imm16 – безпосередньо 16-розрядне значення; imm32 – безпосередньо 32-розрядне значення.

Наприклад розглянемо програму, наведену нижче.

```
1   section .data
2   section .text
3         global main
4   main:
5         mov  AL,  4          ; завантаження в регістр AL
значення 4
6         mov  AH,  1          ; завантаження в регістр AH
значення 1
7         mov  EBX, 0xABCD     ; завантаження в регістр EBX
значення 0xABCD
8         mov  ECX, EAX        ; копіювання з регістра EAX
в регістр ECX
9         mov  EAX, 1          ; код команди виходу в
регістр EAX
10        mov  EBX, 0          ; код статусу виходу в
регістр EBX
11        int  0x80
```

В п'ятому рядку програми у регістр AL завантажується значення 4, чому відповідає значення рядку 14 зневаджувального виводу (для порівняння статус регістрів на початку програми наведено у рядках 7-9). В шостому рядку

програми в регістр AH завантажено значення 1. Враховуючи, що AH – старша частина регістра та AL – молодша, то значення регістра AX стане $100000100_2 = 104_{16} = 260_{10}$ (рядок 21 зневаджувального виводу). В сьомому рядку програми в регістр EBX завантажується значення $ABCD_{16} = 43981$ (рядок виводу 29). Далі значення 104_{16} копіюється з регістра EAX в регістр ECX (рядок виводу 37). Відповідний зневаджувальний вивід наведено нижче.

```
1   (gdb) break main
2   Breakpoint 1 at 0x400080
3   (gdb) run
4   Starting program:
/home/choporoff/Projects/Assembler/3/lab3
5   Breakpoint 1, 0x0000000000400080 in main ()
6   (gdb) info registers
7   rax            0x0   0
8   rbx            0x0   0
9   rcx            0x0   0
10  rip            0x400080   0x400080 <main>
11  (gdb) nexti
12  0x0000000000400082 in main ()
13  (gdb) info registers
14  rax            0x4   4
15  rbx            0x0   0
16  rcx            0x0   0
17  rip            0x400082   0x400082 <main+2>
18  (gdb) nexti
19  0x0000000000400084 in main ()
20  (gdb) info registers
21  rax            0x104260
22  rbx            0x0   0
23  rcx            0x0   0
24  rip            0x400084   0x400084 <main+4>
25  (gdb) nexti
26  0x0000000000400089 in main ()
27  (gdb) info registers
28  rax            0x104260
29  rbx            0xabcd    43981
30  rcx            0x0   0
31  rip            0x400089   0x400089 <main+9>
32  (gdb) nexti
33  0x000000000040008b in main ()
34  (gdb) info registers
35  rax            0x104260
36  rbx            0xabcd    43981
```

```
rcx            0x104260
rip            0x40008b   0x40008b <main+11>
(gdb) nexti
0x0000000000400090 in main ()
(gdb) info registers
rax            0x1  1
rbx            0xabcd     43981
rcx            0x104260
rip            0x400090   0x400090 <main+16>
(gdb) nexti
0x0000000000400095 in main ()
(gdb) info registers
rax            0x1  1
rbx            0x0  0
rcx            0x104260
rip            0x400095   0x400095 <main+21>
(gdb) nexti
[Inferior 1 (process 3069) exited normally]
```

✍ *Завдання для лабораторної роботи*

На базі коду програми, написаної в лабораторній роботі № 1, створити програму, яка виконує наведені нижче дії.

1. У регістр `ECX` завантажує значення `0xABFεEε`, а в регістр `EBX` – `0` (замість `ε` вказати свій номер за академічним журналом у шістнадцятирічній системі числення).

2. Копіює данні з регістра `CH` в регістр `BL`.

3. Копіює данні з регістра `CL` в регістр `BH`.

4. В регістр `EAX` спочатку завантажую значення `0xFFFFFFFF` потім в регістр `AX` – `0`.

Для перегляду стану регістрів використовувати зневаджувач `gdb` (або будь-який інший). Обов'язково необхідно показати зміни регістрів після кожної команди.

Створити звіт, в якому представлено хід роботи та відповіді на контрольні запитання.

Поясніть значення у регістрах `EAX`, `EBX` та `ECX`.

? *Контрольні запитання*

1. Що таке регістр?
2. Які типи регістрів існують на архітектурі `x86`?
3. Порівняйте регістри архітектур `arm` та `x86`?
4. Які особливості регістрів присутні у процесорів архітектури `x84_64`?
5. Поясніть особливості завантаження даних у регістри.

Лабораторна робота № 4. Робота з основною пам'яттю комп'ютера

Теоретичні відомості

Для роботи з операндами, розташованими в основній пам'яті, їх необхідно спочатку оголосити (зв'язати адреси пам'яті з ідентифікаторами для посилання на них). Для цього слугують псевдокомандами визначення та резервування даних. Ці команди необхідно розмістити в сегменті даних (після його декларації: `section .data`)

Псевдокоманди DB, DW та DD – визначення даних

Одна з найбільш розповсюджених псевдокоманд – це псевдокоманда `DB` (`Define Byte`), яка дозволяє визначати однобайтові дані в основній пам'яті. Загальний формат псевдокоманди `DB`:

```
[name] DB initial-values
```

де `name` необов'язковий ідентифікатор[8] – назва адреси в пам'яті, `initial-values` – перелік вихідних значень. Наприклад,

```
1   zero_byte DB 0
2   two_bytes DB 0x00, 0x01
```

В першому рядку визначено один байт основної пам'яті з вихідним значенням `0` та ідентифікатором `zero_byte`. В другому рядку оголошено два послідовних байти з вихідними значеннями `0x00` та `0x01`. Ідентифікатор `two_bytes` вказує на перший з них (`0x00`), для доступу до другого необхідно використовувати конструкцію виду `[zero_byte + 1]`.

Для визначення дані в пам'яті, кратні слову[9], використовується псевдокоманда `DW` (`Define Word`). Формат псевдокоманди `DW` аналогічний до формату команди `DB`. В наступних рядках наведено приклади визначення даних, кратних слову.

```
1   w DW 0x12AB     ;
2   w_array DW 0x1, 0x2, 0x3, 0x4
```

В другому рядку визначено чотири послідовних слова, для доступу до яких необхідно використовувати зміщення відносно адреси першого з них. Наприклад, `[w_array]` – `0x1`, `[w_array + 2]` – `0x2`, тощо. Інструкція

```
MOV AX, [w_array + 4]
```

копіює в регістр `AX` значення, адреса якого на `4` байти більша за адресу `w_array`, то б то, значення `0x3`.

Аналогічно використовуються псевдокоманди `DD`, `DQ` та `DT`, які, відповідно, визначають дані кратні подвійному слову (4 байти), чотирьом словам (8 байт), та 10 байтам.

---

8 Необов'язковість ідентифікатору зумовлена тим, що визначені дані розташовуються у послідовних адресах сегмента даних програми.

9 Історично склалося, що під словом називають два послідовних байти оперативної пам'яті, які використовуються спільно.

Для визначення більш великих масивів даних, що повторюються, можна використовувати псевдокоманду `TIMES`. Наприклад, рядок

```
zeros TIMES 8 DD 0
```

визначає `8` подвійних слів, початкове значення яких `0`.

Розглянемо наступну програму

```
1   section .data                   ; Секція даних
2       my_byte db 7
3       w dw 0x123, 0x124           ; w[0] = 0x123, w[1] =
0x124
4       d dd 0xa, 0xb, 0xc          ; d[0] = 0xa, d[1] =
0xb, d[2] = 0xc
5       chars db 'hello', 10        ; рядок символів
6       zerobuf times 64 db 0       ; 64 байти,
ініціалізовані нулем
7   section .text                   ; Текстова секція (код)
8       global main
9   main:
10      mov  byte [zerobuf + 0], 1    ; zerobuf[0]=1
11      mov  byte [zerobuf + 1], 2    ; zerobuf[1]=2
12      mov  byte [zerobuf + 2], 3    ; zerobuf[2]=3
13      mov  byte AL,  [zerobuf + 2]  ; AL=zerobuf[2]=3
14      mov  byte AH,  [zerobuf + 50] ; AH=zerobuf[50]=0
15      mov  dword     EBX, [d + 8]   ; EBX=d[2]=0xc
16      mov  word CX,  [w + 2]        ; CX=w[1]=0x124
17
18      mov  EAX, 1    ; код команди завершення роботи
програми
19      mov  EBX, 0    ; код статусу завершення
20      int  0x80 ; переривання операціної системи
```

З першого по шостий рядок – секція даних. В рядку 2 визначена змінна `my_byte` з початковим значенням 7. В рядку 3 `w` є масивом з двох слів. В рядку 4 визначено масив `d` з трьох подвійних слів. В наступному рядку визначено 6 байтів – рядок символів. В шостому рядку визначено визначено масив-буфер з 64 байтів, кожен з яких ініціалізовано нулем. Якщо цю програму запустити в зневаджувачі, то отримаємо наступний вивід

```
1   (gdb) break main
2   Breakpoint 1 at 0x8048080
3   (gdb) run
4   Starting program: /home/usergis/4/lab4
5   Breakpoint 1, 0x08048080 in main ()
6   (gdb) nexti
7   0x08048087 in main ()
8   (gdb) nexti
9   0x0804808e in main ()
```

```
(gdb) nexti
0x08048095 in main ()
(gdb) nexti
0x0804809a in main ()
(gdb) nexti
0x080480a0 in main ()
(gdb) nexti
0x080480a6 in main ()
(gdb) nexti
0x080480ad in main ()
(gdb) disassemble
Dump of assembler code for function main:
   0x08048080 <+0>: movb   $0x1,0x80490d3
   0x08048087 <+7>: movb   $0x2,0x80490d4
   0x0804808e <+14>:movb   $0x3,0x80490d5
   0x08048095 <+21>:mov    0x80490d5,%al
   0x0804809a <+26>:mov    0x8049105,%ah
   0x080480a0 <+32>:mov    0x80490c9,%ebx
   0x080480a6 <+38>:mov    0x80490bf,%cx
=> 0x080480ad <+45>:mov    $0x1,%eax
   0x080480b2 <+50>:mov    $0x0,%ebx
   0x080480b7 <+55>:int    $0x80
End of assembler dump.
(gdb) print /x $ax
$1 = 0x3
(gdb) print /x $ebx
$2 = 0xc
(gdb) print /x $cx
$3 = 0x124
(gdb) print /x d@3
$4 = {0xa, 0xb, 0xc}
```

✍ *Завдання для лабораторної роботи*

Розробити програму, яка виконує наступні дії:

1. Визначає в пам’яті рядок символів з ім’ям `my_name`, який містить прізвище та ім’я автора лабораторної роботи.
2. Визначає наступні однобайтові дані `l_byte = N`, `h_byte = 3 N`.
3. Визначає слова `l_word = 10 N`, `h_word = 11 N + 15`.
4. Визначає подвійне слово `v_dword = 30 N + N`^{2}.
5. Копіює визначені дані з пам’яті у регістри.
6. Створює масив з `20` подвійних слів, ініціалізованих `0`.
7. Змінює значення першого, п’ятого, десятого та двадцятого елементів, значеннями, відповідно, `1`, `5`, `10`, `20`.
8. Міняє місцями значення змінної `v_dword` та десятого елементу масиву.

? *Контрольні запитання*

1. Поясніть роботу основних псевдокоманд визначення даних.
2. Яким чином переслати дані з однієї змінної до іншої?
3. Чи можливо копіювання даних з області пам’яті, визначеної псевдокомандою `DW`, в область пам’яті, визначену командою `DD`?
4. Поясніть основні відмінності команд `RESB`, `RESW`, `RESD`, `RESQ` та `REST` від аналогічних `DB`, `DD`, `DQ`, `DT` та `DT`.

✕✎ Яким чином зарезервувати пам’ять для 100 подвійних слів неініціалізованих даних?

Для нотаток

Лабораторна робота № 5. Робота зі стеком. Виклик зовнішніх процедур. Друк на консоль

Теоретичні відомості

Стек

При програмуванні часто виникає необхідність тимчасового збереження вмісту регістрів процесора або деякої адреси пам'яті, щоб потім відновити вихідні значення.

Стек – це тип черги – останній прийшов – перший пішов[10] (рис. 2). Таку чергу можна розглядати як стовпець монет: першу покладену на стіл монету можна забрати лише останньою, якщо монети забирати по одній.

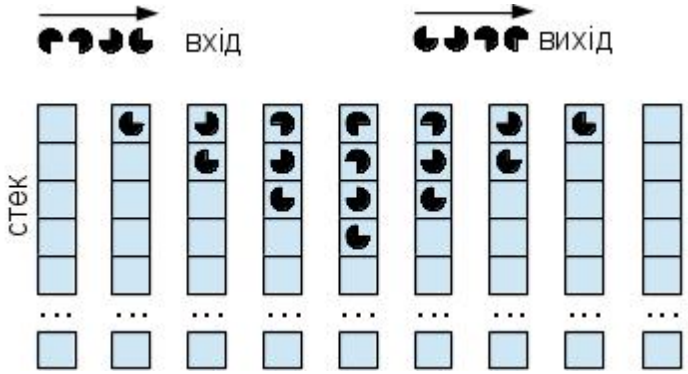


Рис. 2. Модель стека

В PC-сумісних процесорах відсутній апаратний стек, тому дані стека зберігаються в пам'яті. Верхівка стека представлена парою регістрів `SS:ESP` – сегмент стека + вказівник верхівки стека. Будь-яка операція зі стеком має справу з його верхівкою, на яку завжди вказує регістр `ESP`. Стек може зберігати `16-` або `32-`бітні дані.

Команда `push` дозволяє зберегти в стеці дані з будь-якого `16-` або `32-`бітного регістра або області пам'яті. Формат команди `push` наступний:

```
push o1
```

де `o1` – `16-` або `32-`бітний операнд.

Наприклад, команди

```
push EAX
push [variable]
```

спочатку скопіюють у стек значення регістра `EAX`, а потім змінної `[variable]`.

Для вилучення даних зі стека використовується команда `pop`, формат якої аналогічний до формату команди `push`. Наприклад, команди

```
pop [variable]
pop EAX
```

10 LIFO (Last In – First Out).

послідовно вилучають та копіюють дані зі стека: верхівку до змінної [variable], наступне після верхівки значення до регістра EAX.

Для одночасного збереження та відновлення даних всіх регістрів загального призначення використовується пара команд pusha / popa, а для додаткових регістрів команди pushad / popad.

Виклик зовнішніх процедур

Для виклику зовнішніх процедур і функцій з програми на мові асемблера необхідно на початку програми оголосити відповідні процедури (функції) як зовнішні за допомогою команди extern. Наприклад, команда

```
extern printf
```

оголошує зовнішню функцію, назва якої printf.

Програми, які використовують зовнішні процедури (функції) повинні компонуватися з відповідними зовнішніми бібліотеками. Якщо програма використовує зовнішні функції з інших мов програмування, то найбільш простим способом є використання компонувальника з мови програмування, зовнішні функції якої використовуються. Наприклад, у випадку наведеної в прикладі функції printf з мови програмування C більш зручним є використання компонувальника gcc. Наприклад, команда

```
gcc -o lab lab.o
```

компонує об'єктний файл lab.o з стандартною бібліотекою мови C та створює файл для виконання.

Важливим етапом виклику зовнішніх процедур та функцій є передача переліку параметрів. Спосіб передачі параметрів у підпрограми стандартизований для кожної апаратно-програмної платформи. Способи передачі на 32- та 64-бітних платформах відрізняються.

На 32-бітних комп'ютерах аргументи підпрограми необхідно зберегти в стек з порядком від останнього аргументу до першого (з права наліво). Наприклад, наведена нижче програма друкує на екрані значення змінних a, b, c і d. Для управління форматом виводу використовуються рядок fmt.

```
1   extern    printf
2   SECTION .data             ; Секція даних
3        a    dd   5          ; int a=5;
4        b    db   103        ; char b = 103;
5        c    dw   -257       ; short c = -257;
6        d    dq   0x12345678; long d = 0x12345678;
7        fmt  db   "a=%d, b=%hhd, c=%hd, d=%lx", 10, 0 ;
Рядок формату
8   SECTION .text             ; Секція кода
9        global main          ; Точка входу
10  main:
11       push EBP
12       mov  EBP, ESP
13                       ; збереження параметрів у стек
```

```
14      push dword [d]
15      push dword [c]
16      push dword [b]
17      push dword [a]
18      push dword fmt
19      call printf     ; виклик printf(fmt, a, b, c, d);
20      add  ESP, 20    ; у стек було збережено 5
подвійних слів
21                      ; вертаємо вказівник на верхівку стека
22      mov  ESP, EBP
23      pop  EBP
24      mov  EAX, 0
25      ret
```

На 64-бітних системах на базі Unix-подібних операційних систем[11] для передавання параметрів підпрограмам використовуються SSE регістри. Порядок параметрів – з права на ліво. Порядок розміщення в регістрах наступний: RDI, RSI, RDX, RCX, R8, R9, XM0 – XM7. Наведена нижче програма є аналогом попередньої на випадок 64 біт.

```
1   extern    printf
2   SECTION .data         ; Секція даних
3       a dd 5            ; int a=5;
4       b db 103          ; char b = 103;
5       c dw -257         ; short c = -257;
6       d dq 0x12345678; long d = 0x12345678;
7       fmt db "a=%d, b=%hhd, c=%hd, d=%lx", 10, 0 ;
Рядок формату функції printf, "\n",'0'
8   SECTION .text         ; Секція кода
9       global main       ; Стандартна точка старту gcc
10  main:                 ; мітка для точки старту
11      mov  R8, [d]      ; п’ятий аргумент функції printf
12      mov  RCX, [c]     ; четвертий аргумент функції
printf
13      mov  RDX, [b]     ; третій аргумент функції printf
14      mov  RSI, [a]     ; другий аргумент функції printf
15      mov  RDI, fmt     ; перший аргумент функції printf
16      mov  RAX, 0
17      call printf       ; Виклик функції друку
18      mov  RAX, 0
19      mov  EAX, 1
20      mov  EBX, 0
21      int  0x80
```

Виконання обох програм приводить до друку на консоль наведеного нижче повідомлення.

11 GNU/Linux, BSD, Mac OS X (компілятори GCC, Intel C++ Compiler).

```
a=5, b=103, c=-257, d=12345678
```

Завдання для лабораторної роботи

Розробіть програму, яка викликає зовнішню функцію `printf` для друку на консоль змінних, визначених у лабораторній роботі № 4.

? Контрольні запитання

1. Що таке стек?
2. Поясніть роботу команди `PUSH`. Наведіть приклади.
3. Поясніть роботу команди `POP`. Наведіть приклади.
4. Порівняйте способи передачі параметрів на різних платформах.

Для нотаток

Лабораторна робота № 6. Арифметичні операції. Цілі числа без знака

Теоретичні відомості

Розглянута вище команда MOV відноситься до групи команд переміщення даних. У фокусі поточної лабораторної роботи група найчастіше використаних команд – арифметичні операції. Процесор 80386 не містить математичного сопроцесора, тому розглянемо в цій роботі цілочисельну арифметику, яка повністю підтримується процесором 80386. Кожна арифметична команда змінює регістр ознак.

Інструкції додавання add і віднімання sub

Найпростіші арифметичні команди: додавання (add) та віднімання (sub).

Команда add використовує два операнди:

```
add o1, o2
```

Ця команда додає обидва операнди і записує результат в o1.

Аналогічно працює команда віднімання – sub:

```
sub o1, o2
```

Результат операції віднімання, я і у випадку з додавання, записується в o1.

Приклади

```
add AX, 8              ;   AX = AX + 8
add EBX, ECX           ;   EBX = EBX + ECX
sub CX, DX             ;   CX = CX - DX
sub EAX, [var]             ;   EAX = EAX - var
add dword [var4], 123  ;   var4 = var4 + 123; var4
  займає 4 байти
```

В останньому рядку наведеного вище прикладу показано, яким чином можна зазначати об’єм пам’яті, що відповідає операндам.

Інструкції інкременту inc та декременту dec

Інструкція інкременту inc додає до, а декременту dec, відповідно, віднімає від значення операнда одиницю. Ці інструкцію мають однаковий формат. Наприклад,

```
inc EAX            ; EAX = EAX + 1
dec word [number]    ; number = number - 1
```

перша команда збільшує значення регістра EAX, а друга команда зменшує поточне значення змінної number, довжина якої слово, на одиницю.

Інструкція множення mul

Команда mul може бути записана в трьох різних форматах:

```
mul r/m8
mul r/m16
mul r/m32
```

У 8-розрядній формі операнд може бути будь-яким 8-бітним регістром або адресою пам'яті. Другий операнд повинен зберігатися у регістрі AL. Результат (добуток) буде записано у регістр AX.

У 16-розрядній формі (аналогічно то 8-розрядної) операнд може бути будь-яким 16-бітним регістром або адресою пам'яті. Другий операнд повинен зберігатися у регістрі AX. Результат записується в пару регістрів DX:AX.

Аналогічно у 32-розрядній формі другий операнд знаходиться в регістрі EAX. Результат буде записано у пару регістрів EDX:EAX.

В останніх двох випадках, якщо результат вміщується у молодший з пари регістрів (AX або EAX), то значення старшого регістра (DX або EDX) встановлюється рівним 0, інакше в в старший регістр записується частина добутку, яка не вмістилася у молодший регістр.

Інструкція знаходження частки та залишку `div`

Подібно до команди mul, інструкція div може бути представлена в трьох форматах:

```
div r/m8
div r/m16
div r/m32
```

Операнд слугує дільником, а ділене зберігається у фіксованому місці. У 8-бітній формі операнд (дільник) може бути будь-яким 8-бітним регістром або адресою пам'яті. Ділене знаходиться в регістрі AX. Результат: частка – в AL, залишок – в AH.

У 16-бітній формі, аналогічно до 8-бітної, операнд – будь-який 16-бітний регістр або адреса пам'яті, частка зберігається в регістрі AX, залишок – DX.

У 32-бітні формі частка зберігається в регістрі EAX, залишок – EDX.

Застосування арифметичних операцій

Розглянемо приклад застосування арифметичних операцій для знаходження суми, різниці, добутку та відношення двох змінних[12].

```
1   extern    printf
2   SECTION .data        ; Секція даних
3        a    dd   103   ; int a = 103;
4        b    dd   259   ; int b = 259;
5        c    dd   0     ; int c = 0;
6        fmt  db   "a=%d, b=%d, c=%d", 10, 0; Рядок
формату функції printf, "\n",'0'
7        fmt_add   db   "c = a + b = %d", 10, 0  ; Рядок
формату для друку суми
```

12 У наведеному прикладі розроблено програмний код для 64-бітової системи. Еквівалентний приклад на випадок 32 біт наведено у додатку Б.

```
8       fmt_sub   db   "c = a - b = %d", 10, 0   ; Рядок
формату для друку різниці
9       fmt_mul   db   "c = a * b = %d", 10, 0   ; Рядок
формату для друку добутку
10      fmt_div   db   "[a / b] = %d, {a / b} = %d", 10,
0   ; Рядок формату друку для знаходження частки та
залишку
11  SECTION .text        ; Секція коду
12      global main      ; Стандартна точка старту gcc
13  main:                ; мітка для точки старту
14      mov  RCX, [c]
15      mov  RDX, [b]
16      mov  RSI, [a]
17      mov  RDI, fmt
18      mov  RAX, 0
19      call printf      ; Виклик функції для друку
значень a, b, c
20      mov  RAX, 0
21                       ; Обчислення c = a + b
22      mov  EAX, [a]    ; EAX = a
23      add  EAX, [b]    ; EAX = EAX + b = a + b
24      mov  [c], EAX    ; c = EAX = a + b
25                       ; Друк
26      mov  RSI, [c]
27      mov  RDI, fmt_add
28      mov  RAX, 0
29      call printf      ; Виклик функції для друку
30      mov  RAX, 0
31                       ; Обчислення c = b - a
32      mov  EAX, [b]    ; EAX = b
33      sub  EAX, [a]    ; EAX = EAX - a = b - a
34      mov  [c], EAX    ; c = EAX = b - a
35                       ; Друк
36      mov  RSI, [c]
37      mov  RDI, fmt_sub
38      mov  RAX, 0
39      call printf      ; Виклик функції для друку
40      mov  RAX, 0
41                       ; Обчислення c = a * b
42      mov  EAX, [a]    ; EAX = a
43      mul  dword [b]   ; EDX:EAX = EAX * b = a * b
44      mov  [c], EAX    ; c = EAX = a * b
45                       ; Друк
46      mov  RSI, [c]
47      mov  RDI, fmt_mul
```

```
48      mov  RAX, 0
49      call printf     ; Виклик функції для друку
50      mov  RAX, 0
51                      ; Обчислення c = b / a
52      mov  EAX, [b]   ; EAX = b
53      mov  EDX, 0
54      div  dword [a] ; EAX - частка, EDX - залишок
55                      ; Друк
56      mov  EDX, EDX
57      mov  ESI, EAX
58      mov  RDI, fmt_div
59      mov  RAX, 0
60      call printf     ; Виклик функції для друку
61      mov  RAX, 0
62      mov  EAX, 1
63      mov  EBX, 0
64      int  0x80
```

В секції даних (рядки 2-10) оголошено змінні та рядки формату для виводу результатів на консоль. В рядках 22-24, 32-34, 42-44 та 52-54 наведено приклади обчислення суми, різниці, добутку та частки з залишком. В результаті виконання наведеної програми отримаємо наступні рядки на екрані комп'ютера:

```
a=103, b=259, c=0
c = a + b = 362
c = a - b = 156
c = a * b = 26677
[a / b] = 2, {a / b} = 53
```

✍ *Завдання для лабораторної роботи*

Розробити програму для обчислення заданої формули ([ ] – частка від ділення, {} – залишок).

1. $A = 10N - [N/2] + \{N/3\}$.
2. $A = 3N - [N/10] - \{100/N\}$.
3. $A = N^2 - [N/3] + \{2N/3\}$.
4. $A = 10N - [N/2] + \{N/3\}$.
5. $A = N^3 + \{N/3\}$.
6. $A = 10([N/2] + \{N/3\})$.
7. $A = [N/2] + \{(5N+1)/3\}$.
8. $A = 2\{(2N-1)/3\}$.
9. $A = 5[(2N)/3]$.
10. $A = [N/2] \cdot \{N/4\}$.
11. $A = N^2 - N + [4/3]$.
12. $A = 20N - \{N/3\}$.
13. $A = N + [N/4] + \{N/5\}$.

14. $A = 2N - [N/3] + 3\{N/7\}$.

15. $A = 3N - 2[N/7] + 5\{N/8\}$.

Змінна N – подвійне слово. Навести приклади роботи програми для різних значень N з перевіркою.

? *Контрольні запитання*

1. Яким чином обчислити суму значень з основної пам'яті комп'ютера?
2. Яким чином вказати розмір операндів арифметичних операцій?
3. Яким чином підрахувати суму значень всіх регістрів загального призначення?
4. Як знайти залишок від ділення двох значень з основної пам'яті комп'ютера?

✕✎ Наведіть приклад програмного коду для знаходження третьої степені числа, що зберігається у регістрі `EAX`.

🗐 *Для нотаток*

Лабораторна робота № 7. Арифметичні операції. Цілі числа зі знаком

Теоретичні відомості

Розглянуті у попередній лабораторній роботі розглянуті арифметичні операції призначені для роботи з цілими числами без знака (додатними числами). Для роботи з цілими числами зі знаком (діапазоном чисел, який включає від'ємні числа) у мову Асемблер додано спеціальні інструкції.

Команда `neg`

Інструкція `neg` має такий формат:

```
neg r/m8
neg r/m16
neg r/m32
```

Інструкція `neg` має один операнд та змінює його знак на протилежний: додатне число перетворюється у від'ємне та оборотно. Розмір операнда може бути довільним: `8`, `16` або `32` біт.

Наприклад,

```
neg EBX
```

змінює знак значення, яке зберігається у регістрі `EBX`,

```
neg byte   [number]
```

змінює знак однобайтової змінної `number`.

Цілочисельне множення та ділення

Зберігання чисел у додатковому коді робить можливим їх додавання та віднімання зі знаком та без знака за допомогою тих самих інструкцій `add` і `sub`. Проте множення та ділення необхідно використовувати окремі інструкції – `imul` та `idiv`.

Інструкція `imul` помножує цілі числа зі знаком та може мати один, два або три операнди. Коли використовується один операнд поведінка цієї інструкції така сама як інструкції `mul`, проте буде враховуватися знак числа.

Якщо зазначено два операнди, то інструкція `imul` помножить операнди та збереже результат добутку до першого операнда. Тому перший операнд повинен бути регістром. Другий операнд може бути регістром, адресою пам'яті або безпосереднім значенням.

```
imul EAX, EBX ;   EAX = EAX * EBX
imul EDX, [value]    ;   EDX = EDX * [value]
imul EBX, 77  ;   EBX = EBX * 7
```

Якщо зазначено три операнди, то інструкція `imul` знайде їх добуток, а результат збереже до першого операнда. Перший операнд повинен бути регістром, другий може бути будь-якого типу, а третій – безпосереднє значення.

```
imul EAX, [val], 2   ;   EAX = EAX * [val] * 2
imul EDX, EBX, -3    ;   EDX = EDX * EBX * (-3)
```

Інструкція `idiv` використовується для ділення чисел зі знаком, синтаксис її такий самий, як у інструкції `div`.

Наприклад, для обчислення виразу $2a^2 - 7a$, якщо $a = -10$, можна скористатися такою програмою.

```
1   extern    printf
2   SECTION .data        ; Секція даних
3        a    dd   -10   ; int a = -10;
4        f_init    db   "a=%d", 10, 0  ; Рядок формату
для руку вихідних даних
5        f_rslt    db   "c = 2 * a^2 - 7 * a = %d", 10, 0
         ; Рядок формату для друку результату
6   SECTION .text        ; Секція коду
7        global main     ; Стандартна точка старту gcc
8   main:                ; мітка для точки старту
9        push dword [a]
10       push dword f_init
11       call printf
12       add  ESP, 8
13       mov  EAX, [a]   ; EAX = a
14       imul EAX, EAX, 2 ;  EAX = EAX * EAX * 2 = 2 *
a^2
15       mov  EBX, [a]   ; EBX = a
16       imul EBX, 7     ; EBX = EBX * 7 = 7 * a
17       sub  EAX, EBX   ; EAX = EAX - EBX = 2 * a^2 - 7 *
a
18       push dword EAX
19       push dword f_rslt
20       call printf          ; Виклик функції для друку
21       add  ESP, 8
22       mov EAX, 0
23       ret
```

✍ *Завдання для лабораторної роботи*

Розробити програму для обчислення формули ([ ] – частка від ділення, $\{\}$ – залишок).

1. $A = -(10N - [N/2] + \{N/3\} + 8)$.
2. $A = -(3N - [50/10])$.
3. $A = -(N^2 - [N/3])$.
4. $A = -(10N - [N/2] - \{N/3\})$.
5. $A = -(N^3 + \{N/3\})$.
6. $A = -([N/2] + 4 \cdot \{N/3\})$.

7. $A = -(\lfloor N/2 \rfloor + \{(5N+1)/3\})$.
8. $A = -2\{(2N-1)/3\}$.
9. $A = -(\lfloor (2N)/3 \rfloor - 10)$.
10. $A = -(\lfloor N/2 \rfloor \cdot \{N/4\})$.
11. $A = -(N^2 - N + 5)$.
12. $A = -(20N - \{N/3\})$.
13. $A = -(N + \lfloor N/4 \rfloor + \{N/5\})$.
14. $A = -(2N - \lfloor N/3 \rfloor + 3\{N/7\})$.
15. $A = -(3N - 2\lfloor N/7 \rfloor + 5\{N/8\})$.

Змінна N – подвійне слово. Навести приклади роботи програми для різних значень N з перевіркою.

? *Контрольні запитання*

1. Яким чином обчислити суму двох від'ємних значень з основної пам'яті комп'ютера?
2. Яким чином вказати розмір операндів арифметичних операцій?
3. Яким чином підрахувати добуток значень всіх регістрів загального призначення, якщо необхідно врахувати знак?
4. Як знайти залишок від ділення двох значень з основної пам'яті комп'ютера?
5. Наведіть приклад програмного коду для знаходження третьої степені від'ємного числа, що зберігається у регістрі `EAX`.

Для нотаток

Лабораторна робота № 8. Логічні операції

Теоретичні відомості

До логічних операцій відносяться: логічне І ($\wedge$); логічне АБО ($\vee$); АБО, що виключає (xor); логічне заперечення ($\neg$). Інструкції, які реалізують логічні операції, змінюють регістр ознак. Логічні операції виконуються побітово: опрацьовують окремі біти або пари біт (результати логічних операцій наведено у таблиці 1).

Таблиця 1 – Логічні операції

o1	o2	$\neg o1$	$\neg o2$	$o1 \wedge o2$	$o1 \vee o2$	o1 xor o2
0	0	1	1	0	0	0
0	1	1	0	0	1	1
1	0	0	1	0	1	1
1	1	0	0	1	1	0

Інструкції `and`, `or` та `xor`

```
and o1, o2
or  o1, o2
xor o1, o2
```

Інструкція `and` відповідає логічному І та виконує логічне множення двох операндів – `o1` та `o2`. Результат зберігається до операнда `o1`. Типи операндів аналогічні до інструкції `add`. Аналогічно діють інструкції `or` та `xor` – відповідно, логічне АБО та АБО, що виключає.

Інструкція `not`

```
not o
```

Інструкція `not` відповідає логічному запереченню. Ця інструкція має один операнд, над бітами якого виконується логічне заперечення. Результат заперечення до операнду інструкції. Інструкція `not` може використовуватися в одному з трьох форматів:

```
not r/m8
not r/m16
not r/m32
```

Розглянемо приклад програми, що демонструє застосування логічних інструкцій.

```
1   extern    printf
2   SECTION .data        ; Секція даних
3        a    dd   9949h; int a = 39241;
4        b    dd   892Dh; int b = 35117;
```

```
5       c    dd   0          ; int c = 0;
6       fmt  db   "a=%u, b=%u", 10, 0 ; Рядок формату
для руку вихідних даних
7       fmt_and   db   "c = a & b = %u", 10, 0  ; Рядок
формату для друку логічного І
8       fmt_or    db   "c = a | b = %u", 10, 0  ; Рядок
формату для друку логічного АБО
9       fmt_xor db     "c = a xor b = %u", 10, 0; Рядок
формату для друку логічного АБО, що виключає
10      fmt_neg db"neg a = %u, neg b = %u", 10, 0; Рядок
формату друку логічного заперечення
11  SECTION .text        ; Секція коду
12      global main      ; Стандартна точка старту gcc
13  main:                ; мітка для точки старту
14      push dword       [b]
15      push dword       [a]
16      push dword       fmt
17      call printf      ; Виклик функції для друку
значень a і b
18      add  ESP, 12
19                       ; Обчислення c = a & b
20      mov  EAX, [a]    ; EAX = a
21      and  EAX, [b]    ; EAX = EAX & b = a & b
22      mov  [c], EAX    ; c = EAX = a & b
23      push dword       [c]
24      push dword       fmt_and
25      call printf      ; Виклик функції для друку
26      add  ESP, 8
27                       ; Обчислення c = a | b
28      mov  EAX, [a]    ; EAX = a
29      or   EAX, [b]    ; EAX = EAX | b = a | b
30      mov  [c], EAX    ; c = EAX = a | b
31      push dword       [c]
32      push dword       fmt_or
33      call printf      ; Виклик функції для друку
34      add  ESP, 8
35                       ; Обчислення c = a xor b
36      mov  EAX, [a]    ; EAX = a
37      xor  EAX, [b]    ; EAX = EAX xor b = a xor b
38      mov  [c], EAX    ; c = EAX = a xor b
39      push dword       [c]
40      push dword       fmt_xor
41      call printf      ; Виклик функції для друку
42      add  ESP, 8
43                       ; Обчислення заперечення
```

```
44      not  dword     [a]  ; заперечення a
45      not  dword     [b]  ; заперечення b
46      push dword     [b]
47      push dword     [a]
48      push dword     fmt_neg
49      call printf          ; Виклик функції для друку
50      add  ESP, 12
51      mov  EAX, 0
52      ret
```

В секції даних програми (рядки 3-10) оголошені вихідні дані програми. Приклад знаходження логічного І двох змінних наведено в рядках 19-26. Аналогічно в рядках 27-42 знаходяться результати логічного АБО та логічного АБО, що виключає. Потрібно звернути увагу на те, що операція `not` (рядки 44-45) замінює значення операнда результатом його заперечення. Результат роботи цієї програми є наступним:

```
a=39241, b=35117
c = a & b = 35081
c = a | b = 39277
c = a xor b = 4196
neg a = 4294928054, neg b = 4294932178
```

Перевірку логічних операції, які було використано у програмі, наведено у таблиці 2.

Таблиця 2 – Результати логічних операцій над змінними `a` і `b`

	Основа системи числення		
	2	10	16
	Значення		
a	0000 0000 0000 0000 1001 1001 0100 1001	39241	9949
b	0000 0000 0000 0000 1000 1001 0010 1101	35117	892D
a ∧ b	0000 0000 0000 0000 1000 1001 0000 1001	35081	8909
a ∨ b	0000 0000 0000 0000 1001 1001 0110 1101	39277	996D
a xor b	0000 0000 0000 0000 0001 0000 0110 0100	4196	1064

✍ *Завдання для лабораторної роботи*

Розробити програму для обчислення заданої формули. У звіті обов'язково навести перевірку результатів логічних операцій. Вважати змінні – подвійні слова: $a = 2013 \cdot n$, $b = 2014 \cdot n - 10$, $c = 2015 \cdot n + 25$, n – номер у журналі академічної групи.

1. $z = a \wedge (b \vee c)$.

2. $z = (a \text{ xor } b) \vee c$.
3. $z = a \wedge (\neg b \vee c)$.
4. $z = \neg a \wedge b \wedge c$.
5. $z = a \wedge \neg b \wedge c$.
6. $z = a \wedge b \wedge \neg c$.
7. $z = \neg(a \wedge b) \wedge c$.
8. $z = a \wedge \neg(b \wedge c)$.
9. $z = \neg(a \wedge c) \wedge b$.
10. $z = \neg(a \vee b \vee c)$.
11. $z = a \text{ xor } b \text{ xor } c$.
12. $z = \neg(a \wedge b) \vee \neg c$.
13. $z = \neg(a \vee c) \wedge \neg b$.
14. $z = (a \wedge b) \vee \neg c$.

☞✕✎ $z = (a \vee b) \wedge (b \vee c)$.

? *Контрольні запитання*

1. Яким чином знайти результат логічної операції над двома операндами, що зберігаються у пам'яті комп'ютера?
2. Яким чином знайти логічне заперечення виразу?
3. Наведіть приклад знаходження логічної суми п'яти значень.

✔✎ Порівняйте дію інструкцій `or` та `xor`.

🗐 *Для нотаток*

Лабораторна робота № 9. Команди умовного та безумовного переходу

Теоретичні відомості

У мовах високого рівня конструкції розгалуження програми відомі як оператор `IF-THEN`. Такі конструкції дозволяють обрати один з альтернативних варіантів виконання програми. У мові асемблеру механізм вибору реалізовано на базі інструкцій порівняння, умовного та безумовного переходу.

Команди `cmp` і `test`

Команди `cmp` і `test` використовуються для порівняння двох операндів. Операнди повинні мати однаковий розмір.

```
cmp o1, o2
```

Команда `cmp`[13] діє подібно до інструкції `sub`: операнд `o2` віднімається від `o1`. Результат ніде не зберігається, інструкція змінює тільки регістр ознак. Ця інструкція може бути використана як для порівняння цілих беззнакових чисел, так і для порівняння чисел зі знаком.

Команда `test` діє подібно до `cmp`, але замість віднімання вона обчислює порозрядне І операндів. Результат інструкції – змінені біти регістра ознак. Цю інструкцію зручно застосовувати для порівняння бітових масивів.

Наприклад, інструкція

```
cmp AL, 5
```

порівнює значення регістра `AL` зі значенням 5, а інструкція

```
cmp [var], EAX
```

порівнює значення змінної `var` зі значенням регістра `EAX`. Для перевірки значення п'ятого біту регістра `BX` можна скористатися інструкцією

```
test   BX, 00010000b
```
[14]

Команда безумовного переходу

Інструкція безумовного переходу `jmp` дозволяє змінити послідовність виконання програми шляхом завантаження інструкції за вказаною адресою. Ця інструкція фактично змінює стан вказівника команд (регістрів `EIP` та/або `CS`), що «перемикає» процесор на виконання інструкції за потрібною адресою пам'яті. Формат інструкції `jmp` наступний:

```
jmp [type] o
```

Аналогом інструкції `jmp` у мовах високого рівня є конструкції `GOTO`. Операнд `o` може бути зазначений безпосередньо або бути регістром загального призначення, в якому збережено адреса[15]. Також операндом може бути мітка програми для автоматичного обчислення адреси наступної за міткою команди.

---

13 Скорочення від «compare» – «порівняти»

14 Логічне І двійкового значення 10000 з будь-яким числом буде дорівнювати нулю тільки коли п'ятий біт цього числа дорівнює нулю

15 Зазначення безпосередньої адреси команди або її обчислення у регістрах значно ускладнює перевірку кода, тому на практиці не рекомендується використання цих режимів.

Перехід буває трьох типів: короткий (`short`), близький (`near`) або дальній (`far`). Тип переходу – необов'язковий параметр. За замовченням вважається, що перехід типу `near`.

Максимальна довжина[16] переходу обмежена значенням з `8` розрядів (тобто різниця між адресами може бути у межах від `-128` до `127`). Довжина ближнього переходу для `32`-бітних машин у захищеному режимі обмежується 4 Гб адресного простору. Дальній перехід змінює одночасно лічильник команд на сегмент коду, що дозволяє вийти за адресний простір програми[17].

Міткою у програму може бути будь-яка послідовність символів, яка нерезервована для службових слів асемблера або під імена змінних. Мітка обов'язково завершується двокрапкою, після якої зазначається послідовність команд, що буде виконуватись. Наприклад, нескінченний цикл може мати такий вигляд:

```
…
loop_begin:
add EAX, EBX
inc ECX
jmp loop_begin
…
```

Умовні переходи

Інструкції умовного переходу базуються на перевірці стану бітів регістра ознак, за результатом якої виконується або перехід до команди за визначеною адресою, або до наступної команди.

Команди умовного переходу мають спільний формат:

```
jx label
```

Як правило, командам умовного переходу передує інструкція порівняння операндів `cmp` або `test`. Найбільш поширені інструкції умовного переходу наведено у таблиці 3.

Таблиця 3 – Інструкції умовного переходу

Умова	o1=o2	o1≠o2	o1>o2	o1<o2	o1≥o2	o1≤o2
Інструкції для беззнакових чисел	je	jne	ja	jb	jae	jbe
Інструкції для чисел зі знаком	je	jne	jg	jl	jge	jle

16 Під довжиною переходу розуміється різниця між адресами команд при переході

17 Вихід за адресний простір програми може призвести до потрапляння в адресний простір іншої програми та, як наслідок, до нестабільної роботи системи

Наприклад, для обчислення формули $c = \begin{cases} x + a, & a \geq 0, \\ 2x - a, & a < 0 \end{cases}$ можна скористатися такою програмою:

```
1   extern    printf
2   SECTION .data
3         a    dd   -10
4         x    dd   15
5         c    dd   0
6         f_init    db   "x=%d, a=%d", 10, 0
7         f_rslt    db   "c = %d", 10, 0
8   SECTION .text
9         global main
10
11  main:
12        mov  RDX, [a]
13        mov  RSI, [x]
14        mov  RDI, f_init
15        mov  RAX, 0
16        call printf
17        mov  RAX, 0
18        mov  EAX, [x]
19        cmp  dword [a], 0
20        jge  a_gr_eq_0
21        ; перехід до наступної команди відбудеться
тільки у випадку, коли a < 0
22        imul EAX, 2    ; eax = 2 x
23        sub  EAX, [a]  ; eax = 2 x - a
24        jmp rslt;
25        a_gr_eq_0:
26        add  EAX, [a]  ; eax = x + a
27        rslt:
28        mov  [c], EAX
29        mov  RSI, [c]
30        mov  RDI, f_rslt
31        mov  RAX, 0
32        call printf
33        mov  RAX, 0
34        mov  EAX, 1
35        mov  EBX, 0
36        int  0x80
```

У рядку 18 програми виконується порівняння значення `a` з нулем. У рядку 20 інструкція умовного переходу `jge`: якщо `a ≥ 0`, то виконується інструкція за міткою `a_gr_eq_0`, інакше наступна інструкція (рядок 22). У рядку 24

інструкція безумовного переходу для ігнорування рядків 25-26 на випадок, якщо `a < 0`. При `a = -10` та `x = 15` отримано такий результат:

```
x=15, a=-10
c = 40
```

✍ *Завдання для лабораторної роботи*

Розробити програму для обчислення умовної формули.

1. $f(x,y,a)=\begin{cases} x+y, & a<0,\\ 2x, & 0\le a<2,\\ 3y-x, & a\ge 2. \end{cases}$

2. $f(x,y,a)=\begin{cases} x-y, & a\le -1,\\ 2x-y, & -1<a<2,\\ -3y+x, & a\ge 2. \end{cases}$

3. $f(x,y,a)=\begin{cases} x-3y, & a<2,\\ x, & 2\le a<4,\\ y-2x, & a\ge 4. \end{cases}$

4. $f(x,y,a)=\begin{cases} x^2, & a<-2,\\ x-2, & -2\le a<0,\\ y-x+1, & a\ge 0. \end{cases}$

5. $f(x,y,a)=\begin{cases} 2x+y, & a<3,\\ x-y, & 3\le a<5,\\ 3y, & a\ge 5. \end{cases}$

6. $f(x,y,a)=\begin{cases} x+2y, & a<1,\\ x^2-y, & 1\le a<5,\\ y-x, & a\ge 5. \end{cases}$

7. $f(x,y,a)=\begin{cases} x-y, & a<-1,\\ 2x-2y, & -1\le a<2,\\ 3y-3x, & a\ge 2. \end{cases}$

8. $f(x,y,a)=\begin{cases} x+3y, & a<0,\\ 2x-3y, & 0\le a<3,\\ 3y-2x, & a\ge 3. \end{cases}$

9. $f(x,y,a)=\begin{cases} 2x+y, & a<-1,\\ x^2-y, & -1\le a<2,\\ y^2-x, & a\ge 2. \end{cases}$

10. $f(x,y,a)=\begin{cases} x-y, & a<-2,\\ 2x-3y, & -2\le a<2,\\ y-2x, & a\ge 2. \end{cases}$

11. $f(x,y,a)=\begin{cases} x+y-3, & a<0,\\ 2x-a, & 0\le a<3,\\ 3y-a+x, & a\ge 3. \end{cases}$

12. $f(x,y,a)=\begin{cases} x-y+a, & a<0,\\ 2x-2a, & 0\le a<4,\\ 3y-x+a, & a\ge 4. \end{cases}$

13. $f(x,y,a)=\begin{cases} x-y-a, & a<0,\\ x+y-a, & 0\le a<5,\\ y-x-a, & a\ge 5. \end{cases}$

14. $f(x,y,a)=\begin{cases} x+ay, & a<0,\\ 2x-3a, & 0\le a<2,\\ 3y-ax, & a\ge 2. \end{cases}$

15. $f(x,y,a)=\begin{cases} ax+y, & a<0,\\ 2ax, & 0\le a<4,\\ 3ya-ax, & a\ge 4. \end{cases}$

Програма повинна виводити результат обчислення на екран. Змінні *a*, *x* та *y* повинні бути визначені у основній пам'яті (подвійні слова). Навести приклади роботи програми для різних значень змінних.

? *Контрольні запитання*

1. Поясніть роботу команди `TEST`.
2. Якій дії відповідає команда `JMP`?
3. Поясніть роботу команди `CMP`.
4. Наведіть приклади використання групи команд `J*` для організації циклів з передумовою.
5. Наведіть приклади використання групи команд `J*` для організації циклів з постумовою.

Для нотаток

Лабораторна робота № 10. Організація циклів

Теоретичні відомості

Родина x86-сумісних чипів використовує архітектуру CISC, тобто мають повну систему команд. Іншими словами, у складі системи команд маються складні команди, які можуть заміняти певні послідовності простих. На CISC-процесорах не потрібно реалізовувати цикли на базі команд умовного переходу – для цього можна використовувати команду loop:

```
loop label
```

Подібно до команди mul, інструкція loop використовує два операнда. Перший операнд фіксований, який не можливо зазначати явним чином. Це значення регістра ECX (або CX, або RCX). Другий операнд – адреса мітки – початку циклу. Інструкція loop зменшує значення регістра ECX на одиницю і, якщо значення в регістрі ECX не рівне 0, то виконується перехід на мітку label. Мітка повинна бути у межах 128 байтів – короткий перехід.

Наприклад, розглянемо програму для обчислення факторіалу.

```
1   extern    printf
2   SECTION .data
3        n    dd   5
4        frmt db   "%d!=%d", 10, 0
5   SECTION .text
6        global main
7   main:
8        mov  ECX, [n]   ; ECX = n
9        mov  EAX, 1     ; EAX = 1
10       f_begin:
11            mul  ECX   ; EAX = EAX * ECX
12       loop f_begin
13       push EAX
14       push dword [n]
15       push dword frmt
16       call printf
17       add  ESP, 12
18       mov  EAX, 0
19       ret
```

У рядках 10-12 створено цикл, який поточне значення регістра EAX на поточне значення регістра ECX, що зменшується на одиницю після кожної ітерації. Як тільки значення у регістрі ECX буде дорівнювати 0, керування буде передане наступні після loop інструкції – почнеться підготування друку результату. Результат роботи програми матиме такий вигляд:

```
5!=120
```

✍ *Завдання для лабораторної роботи*

Розробити програму для обчислення функції в $n \times m$ цілих точках інтервалу $[a; b] \times [c; d]$ ($n = b - a$, $m = d - c$).

1. $f(x,y) = 2x^2 - 3y,\quad a = 0,\quad b = 5,\quad c = -1,\quad d = 4.$
2. $f(x,y) = 3x^2 - 4y,\quad a = -1,\quad b = 4,\quad c = 2,\quad d = 6.$
3. $f(x,y) = 2x^2 - y - 1,\quad a = 0,\quad b = 10,\quad c = -11,\quad d = 0.$
4. $f(x,y) = x^3 - y^{3,}\quad a = 0,\quad b = 7,\quad c = -1,\quad d = 5.$
5. $f(x,y) = 3x - 3y,\quad a = 0,\quad b = 15,\quad c = 0,\quad d = 15.$
6. $f(x,y) = 2x^2 - 3y^{2,}\quad a = -10,\quad b = 0,\quad c = -10,\quad d = 0.$
7. $f(x,y) = 2x^3 + 3y - 10,\quad a = 0,\quad b = 7,\quad c = -5,\quad d = 5.$
8. $f(x,y) = x^2 + y,\quad a = -5,\quad b = 5,\quad c = -5,\quad d = 5.$
9. $f(x,y) = y - x,\quad a = 0,\quad b = 10,\quad c = -10,\quad d = 0.$
10. $f(x,y) = 3y - x^{2,}\quad a = 0,\quad b = 5,\quad c = -5,\quad d = 0.$
11. $f(x,y) = 3y^2 - x^{2,}\quad a = -5,\quad b = 5,\quad c = -10,\quad d = 0.$
12. $f(x,y) = x^2 - y^{3,}\quad a = -3,\quad b = 3,\quad c = -3,\quad d = 3.$
13. $f(x,y) = 3y - 5x,\quad a = -4,\quad b = 3,\quad c = -4,\quad d = 3.$
14. $f(x,y) = x^2 - 5y,\quad a = 0,\quad b = 7,\quad c = 0,\quad d = 7.$
15. $f(x,y) = 3y - x^{3,}\quad a = 0,\quad b = 5,\quad c = -1,\quad d = 4.$

Програма повинна виводити результати обчислення на екран. Змінні *a*, *b*, *c*, *d*, *x* та *y* повинні бути визначені в основній пам'яті (подвійні слова). Навести приклади роботи програми для різних значень змінних.

? *Контрольні запитання*

1. Яким чином організувати цикли на базі групи команд `Jx` (наведіть приклади)?
2. Поясніть роботу команди `LOOP`.
3. Поясніть роботу команди `LOOPZ`.
4. Поясніть роботу команди `LOOPNZ`.

🗐 *Для нотаток*

Лабораторна робота № 11. Створення підпрограм

Теоретичні відомості

Під час програмування часто необхідно використовувати підпрограми. Основне призначення підпрограм – скорочення кода основної програми: деякі послідовності інструкцій, що повторюються, можна об'єднувати у підпрограми та викликати їх за необхідністю.

Для виклику підпрограм у мові асемблера використовується інструкція `call`, для повернення з підпрограми в основну програму – `ret`. Формат цих інструкцій такий:

```
call [type] label
…
ret
```

Інструкція `call` має два операнди: необов'язковий – тип переходу `(type)` і обов'язковий – адреса підпрограми. Адреса підпрограми може бути зазначена безпосереднім значенням, обчислена в регістрі або пам'яті, міткою. При виконанні інструкції `call` у першу чергу в стеці зберігається значення регістра `EIP`. Потім керування передається інструкції по зазначеній адресі.

Повернення з підпрограми виконується за допомогою інструкції `ret`. Інструкція `ret` «виштовкує» зі стеку його верхівку (значення, що було збережено при виконанні інструкції `call`) в регістр `EIP`. Далі процесор продовжить виконання наступної за `call` інструкції основної програми.

Якщо підпрограма викликалась інструкцією `call far`, то для коректного повернення з такої підпрограми потрібно відновити не тільки значення регістра `EIP`, а ще й регістра `CS`. У цьому випадку потрібно використовувати інструкцію `retf`.

Наприклад, розглянемо програму для обчислення послідовності значень функції $f(i) = i^2 - 7,\ i = \overline{-5,5}$.

```
1   extern    printf
2   SECTION .data
3         n    dd   11
4         frmt db   "%d ^ 2 - 7 = %d", 10, 0
5   SECTION .text
6         global main
7   main:
8         mov  ECX, [n]   ; ECX = n
9         mov  EAX, -5    ; EAX = 1
10        f_begin:
11              call func_
12              pusha     ; зберігаємо значення регістрів
для коректного відновлення після друку
13              push EBX
```

```
14          push dword EAX
15          push dword frmt
16          call printf
17          add  ESP, 12
18          popa        ; відновлення значення регістрів
19          inc  EAX
20      loop f_begin
21      mov  EAX, 0
22      ret
23  func_:              ; функція для обчислення EAX^2-7
24      mov  EBX, EAX
25      imul EBX, EBX
26      sub  EBX, 7
27      ret
```

У рядку 11 викликається підпрограма `func_`, реалізація якої знаходиться в рядках 24-27. Результат роботи програми буде таким:

```
-5 ^ 2 - 7 = 18
-4 ^ 2 - 7 = 9
-3 ^ 2 - 7 = 2
-2 ^ 2 - 7 = -3
-1 ^ 2 - 7 = -6
0 ^ 2 - 7 = -7
1 ^ 2 - 7 = -6
2 ^ 2 - 7 = -3
3 ^ 2 - 7 = 2
4 ^ 2 - 7 = 9
5 ^ 2 - 7 = 18
```

✍ *Завдання для лабораторної роботи*

Розробити програму, яка обчислює задану функцію за допомогою підпрограми. Необхідно що б виклик підпрограми відбувався у циклі при кожній ітерації для обчислення функції в $n \times m$ цілих точках інтервалу $[a; b] \times [c; d]$ ($n = b - a, m = d - c$).

1. $f(x,y) = 2x^2 - 3y,\quad a = 0,\quad b = 5,\quad c = -1,\quad d = 4.$
2. $f(x,y) = 3x^2 - 4y,\quad a = -1,\quad b = 4,\quad c = 2,\quad d = 6.$
3. $f(x,y) = 2x^2 - y - 1,\quad a = 0,\quad b = 10,\quad c = -11,\quad d = 0.$
4. $f(x,y) = x^3 - y^3,\quad a = 0,\quad b = 7,\quad c = -1,\quad d = 5.$
5. $f(x,y) = 3x - 3y,\quad a = 0,\quad b = 15,\quad c = 0,\quad d = 15.$
6. $f(x,y) = 2x^2 - 3y^2,\quad a = -10,\quad b = 0,\quad c = -10,\quad d = 0.$
7. $f(x,y) = 2x^3 + 3y - 10,\quad a = 0,\quad b = 7,\quad c = -5,\quad d = 5.$
8. $f(x,y) = x^2 + y,\quad a = -5,\quad b = 5,\quad c = -5,\quad d = 5.$
9. $f(x,y) = y - x,\quad a = 0,\quad b = 10,\quad c = -10,\quad d = 0.$
10. $f(x,y) = 3y - x^2,\quad a = 0,\quad b = 5,\quad c = -5,\quad d = 0.$

11. $f(x,y)=3y^2-x^2,\quad a=-5,\quad b=5,\quad c=-10,\quad d=0.$
12. $f(x,y)=x^2-y^3,\quad a=-3,\quad b=3,\quad c=-3,\quad d=3.$
13. $f(x,y)=3y-5x,\quad a=-4,\quad b=3,\quad c=-4,\quad d=3.$
14. $f(x,y)=x^2-5y,\quad a=0,\quad b=7,\quad c=0,\quad d=7.$
15. $f(x,y)=3y-x^3,\quad a=0,\quad b=5,\quad c=-1,\quad d=4.$

Програма повинна виводити результати обчислення на екран. Змінні *a*, *b*, *c*, *d*, *x* та *y* повинні бути визначені в основній пам'яті (подвійні слова). Навести приклади роботи програми для різних значень змінних.

? *Контрольні запитання*

1. Яким чином організувати цикли на базі групи команд `Jx` (наведіть приклади)?
2. Поясніть роботу інструкції `loop`.
3. Поясніть роботу інструкції `loopz`.
4. Поясніть роботу інструкції `loopnz`.
5. Поясніть роботу інструкції `call`.

✖✎ Поясніть особливості використання інструкції `ret`.

Для нотаток

Лабораторна робота № 12. Додавання асемблерного коду в програми, які написані мовою C

Теоретичні відомості

Додавання асемблерного коду в програми, які написані на мовах високого рівня, використовується для створення фрагментів, які не аналізуються компілятором, проте програміст визначає яким чином ці додавання взаємодіють з іншими фрагментами програми. Асемблерні додавання на практиці використовуються для програмування специфічних для певного обладнання фрагментів алгоритмів. «Ручне» програмування додавань мовою асемблера дозволяє програмісту керувати ефективністю використання регістрів процесора.

Необхідно пам'ятати, що при компіляції `C`-програми компілятором `gcc` код асемблерного додавання буде транслюватися транслятором `gas`[18], а не `nasm`. Синтаксисом `gas` є `AT&T`[19], який суттєво відрізняються від синтаксису `Intel` (`nasm`). Сучасні версії `gas` підтримують синтаксис `Intel`, активізувати його можна за допомогою спеціальної директиви. Але діалект синтаксису, що підтримується буде трохи відрізнятися від діалекту, що використовується `nasm`.

Додавання асемблерного коду в програми мовою `C` має такий синтаксис:

```
__asm__ (додавання: перелік_вихідних_параметрів:
перелік_вхідних_параметрів: перелік_регістрів);
```

Починається оператор асемблерного додавання з ключового слова `asm` або `__asm__`, за яким у дужках розміщується його реалізація. В коді реалізації додавання можуть біти використані не тільки інструкції асемблера, а і будь-які директиви, що підтримує `gas`. Однією з таких директив є директива зміни синтаксису інструкцій:

```
.intel_syntax noprefix
```

Додавання являє собою рядкову константу з асемблерними інструкціями. Кожна нова інструкція від попередньої повинна бути відокремлена ескейп-послідовністю `\n\t`.

Перелік вихідних параметрів починається з рядка `"=r"`, за яким в дужках через кому перелічуються імена змінних, що зберігатимуть результати роботи асемблерного додавання.

Перелік вхідних параметрів починається рядком `"r"`, за яким у дужках перелічуються імена змінних, що містять вхідні значення.

Буква `"r"` у переліках вхідних та вихідних параметрів вказує на те, що значення між додаванням та `C`-програмою обмінюються через регістри загального призначення. Також можливо вказати який саме з регістрів буде використано для обміну даними (символьні коди та відповідні до них регістри наведені у таблиці 4).

18 GNU асемблер

19 http://ru.wikibooks.org/wiki/Ассемлер_в_Linux_для_программистов_C

Таблиця 4 – Коди регістрів для обміну даних

Символьний код	Регістри
a	eax, ax, al
b	ebx, bx, bl
c	ecx, cx, cl
d	esi, si
S	ebx, bx, bl
D	edi, di

Розмір регістра, що буде використано для обміну даних буде обрано компілятором автоматично, залежно від типу даних змінної, з якою відбувається взаємодія.

Вхідні та вихідні параметри мають наскрізну нумерацію починаючи з нуля. Нумерація виконується з ліва на право з початку вихідних параметрів далі вхідних. Для посилання на значення параметра необхідно використовувати символ `%` (процент). Наприклад, `%1` – посилання на параметр з номером `1`.

Використання міток та посилання на них в асемблерних додаваннях має певні особливості. Кожна мітка є цілим числом, за яким розміщується вертикальна двокрапка. Для посилання на мітки в інструкціях безумовного або умовного переходу необхідно вказати номер мітки з суфіксом, який визначає напрям пошуку мітки. Якщо мітка була оголошена раніше посилання на неї, то використовується суфікс `b`, інакше – `f`. Наприклад, `0b` – посилання на мітку, що оголошена за текстом додавання раніше, `1f` – посилання на мітку, що оголошено для інструкції, розташованих нижче за текстом додавання.

Розглянемо приклад використання асемблерного додавання для обчислення факторіалу числа.

```
1   #include <stdio.h>
2   int factorial(int n)
3   {
4        int result;
5        __asm__( ".intel_syntax noprefix\n\t"
6             "mov ecx, %1\n\t"
7             "mov eax, 1\n\t"
8             "0:\n\t"
9             "mul ecx\n\t"
10            "loop 0b\n\t"
11            "mov %0, eax\n\t"
12            : "=r"(result)
13            : "r"(n)
14            : "%eax", "%ecx", "%edx"
15            );
```

```
16      return result;
17  }
18  int main()
19  {
20      printf("f = %d\n", factorial(3));
21      return 0;
22  }
```

Інструкції асемблерного додавання знаходяться в рядках 5-15. У першому рядку асемблерного додавання оголошено, що буде використано синтаксис `Intel`. У рядку № 6 значення параметру `%1` (вхідний параметр) завантажується в регістр `ECX`. Далі в рядку 7 регістр `EAX` ініціалізується одиницею. Рядок 8 містить оголошення мітки з номером `0`. У рядку 10 інструкція циклу з посиланням на раніше оголошену мітку. Значення з регістра `EAX` завантажується у параметр `%0` (вихідний параметр) в рядку 11. У рядку 12 оголошено, що вихідним параметром є змінна `result`, а в рядку 13, що вхідним – параметр-змінна `n`. Рядок 14 містить перелік регістрів, значення яких буде змінено протягом роботи додавання.

Для компіляції наведеного прикладу необхідно використати таку команду:

```
gcc -std=c99 -Wall -O2 -masm=intel asm_inline.c -
  oasm_inline
```

де `asm_inline.c` – ім'я файлу з текстом програми; `asm_inline` – ім'я файлу для запису результату компіляції.

Якщо виконати команду

```
gcc -std=c99 -Wall -O0 -masm=intel -S asm_inline.c
```

буде компілятором буде згенеровано код програми мовою асемблера (синтаксис `Intel`), що відповідаю програмі, написаній мовою `C` (відповідна до розглянутого прикладу програма наведена у додатку В).

✍ *Завдання для лабораторної роботи*

Розробити програму мовою `C` з асемблерним додаванням для обчислення значення функції. Додавання розмістити в тілі окремої функції. За допомогою розробленої підпрограми обчислити значення функції в $n \times m$ цілих точках інтервалу $[a; b] \times [c; d]$ ($n = b - a$, $m = d - c$).

1. $f(x,y) = 2x^2 - 3y,\quad a = 0,\quad b = 5,\quad c = -1,\quad d = 4.$
2. $f(x,y) = 3x^2 - 4y,\quad a = -1,\quad b = 4,\quad c = 2,\quad d = 6.$
3. $f(x,y) = 2x^2 - y - 1,\quad a = 0,\quad b = 10,\quad c = -11,\quad d = 0.$
4. $f(x,y) = x^3 - y^3,\quad a = 0,\quad b = 7,\quad c = -1,\quad d = 5.$
5. $f(x,y) = 3x - 3y,\quad a = 0,\quad b = 15,\quad c = 0,\quad d = 15.$
6. $f(x,y) = 2x^2 - 3y^2,\quad a = -10,\quad b = 0,\quad c = -10,\quad d = 0.$
7. $f(x,y) = 2x^3 + 3y - 10,\quad a = 0,\quad b = 7,\quad c = -5,\quad d = 5.$
8. $f(x,y) = x^2 + y,\quad a = -5,\quad b = 5,\quad c = -5,\quad d = 5.$

9. $f(x,y)=y-x,\quad a=0,\quad b=10,\quad c=-10,\quad d=0.$
10. $f(x,y)=3y-x^2,\quad a=0,\quad b=5,\quad c=-5,\quad d=0.$
11. $f(x,y)=3y^2-x^2,\quad a=-5,\quad b=5,\quad c=-10,\quad d=0.$
12. $f(x,y)=x^2-y^3,\quad a=-3,\quad b=3,\quad c=-3,\quad d=3.$
13. $f(x,y)=3y-5x,\quad a=-4,\quad b=3,\quad c=-4,\quad d=3.$
14. $f(x,y)=x^2-5y,\quad a=0,\quad b=7,\quad c=0,\quad d=7.$
15. $f(x,y)=3y-x^3,\quad a=0,\quad b=5,\quad c=-1,\quad d=4.$

Програма повинна виводити результати обчислення на екран. Змінні *a*, *b*, *c*, *d*, *x* та *y* повинні бути визначені в основній пам'яті (подвійні слова). Навести приклади роботи програми для різних значень змінних.

? *Контрольні запитання*

1. Поясніть основні принципи використання асемблерних додавань.
2. Якщо оголошено такі параметри: `:"=r"(a) :"r"(b,c,d)`, яким чином завантажити значення змінної `c` у регістр `ECX`?
3. Поясніть основні принципи використання оголошення та посилання на мітки у додаваннях.
4. Якщо у додаванні використано інструкції `jz 1f`, де повинна бути оголошена мітка, з яким номером?
5. Чи можливо використання міток з однаковими номерами у різних частинах програми?

✖✎ Назвіть основні переваги та недоліки використання асемблерних додавань.

🗐 *Для нотаток*

Індивідуальне завдання

Теоретичні відомості

У висвітленому вище курсі лабораторних робіт всі програми призначалися для обробки символьної інформації та виконання операцій на множиною цілих чисел. Для того щоб виконати операцію над дійсним числом, що має цілу і дрібну частину, необхідно використовувати арифметичний процесор. Арифметичні процесори, які також називаються сопроцесорами, оброблюють інформацію, що представлена у форматі дійсних чисел. Сопроцесор дозволяє значно пришвидшити роботу програм, що виконують розрахунки з високою точністю, тригонометричні обчислення та обробку інформації, яку необхідно подати у вигляді дійсних чисел.

Починаючи з моделі процесора `i486DX` арифметичний сопроцесор міститься на одному кристалі з основним процесором. Проте, не зважуючи на цей факт, логічно їх потрібно відокремлювати.

Програмна модель арифметичного сопроцесора – стек регістрів (рис. 3), що накладає певні особливості його застосування.

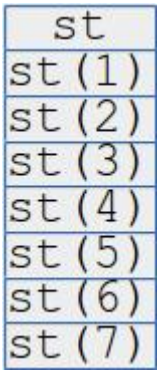


Рис. 3. Програмна модель арифметичного сопроцесора

Програмісту в сопроцесорі доступні 8 регістрів загального призначення, які позначаються `st(0)`, `st(1)`, …, `st(7)`, і п’ять нечислових регістрів. Кожен з регістрів загального призначення має розмір `10` байт. Ці регістри призначені для зберігання операндів і результатів арифметичних операцій з дійсними числами. Перший з цих регістрів – `st(0)`, використовується найбільш часто, тому доступ до нього можна отримати й за скороченою назвою `st`.

Незалежно від формату дані, що завантажуються у сопроцесор, перетворюються у внутрішнє подання, яке займає `80` біт, і записуються в один з регістрів загального призначення. По завершенню обробки результат після попереднього перетворення у необхідний формат може бути пересланим у пам’ять. Основне призначення нечислових регістрів – керування роботою сопроцесора.

Для використання сопроцесора необхідно додати до програми спеціальні інструкції – команди сопроцесора.

Ініціалізація арифметичного сопроцесора виконується за допомогою інструкції `finit`, яка не має жодного параметра. Ця інструкція переводить сопроцесор в його стан за замовченням.

Для обміну даними між основною пам'яттю та сопроцесором використовуються інструкції

```
fild memory
fld memory
fistp memory
fstp memory
```

Інструкції `fild` та `fld` слугують для завантаження з пам'яті у сопроцесор відповідно цілих та дійсних значень. Власне `fistp` та `fst` аналогічно для завантаження зі сопроцесора в основну пам'ять. При цьому відбувається автоматична конвертація подання цілих чисел у подання дійсних чисел та зворотно (при використанні інструкцій `fild` і `fistp`). Для обміну даними можна використовувати змінні з розміром `32`, `64` або `80` біт. Наприклад, наступна програма конвертує дійсне число у ціле.

```
1   extern    printf
2   SECTION .data
3         x    dd   3.14 ; x = 3.14
4         y    dd 0
5         frmt db   "y = %d", 10, 0
6   SECTION .text
7         global main
8   main:
9         fld dword [x]   ; x → st(0)
10        fistp dword [y]; st(0) → y
11        push dword [y]
12        push dword frmt
13        call printf
14        add  ESP, 8
15        mov  EAX, 0
16        ret
```

У рядку 10 дійсне подання числа `3.14` завантажується до сопроцесора на вершину стека. Далі в рядку при копіюванні значення копіюванні верхівки стека в основну пам'ять у рядку 11 це значення конвертується в ціле.

Для додавання дійсних чисел необхідно використовувати інструкцію `fadd`, параметром якої може бути комірка пам'яті або один з регістрів сопроцесора. Результат додавання записується у регістр `ST0`. Наприклад, інструкція

```
fadd   ST1
```

додає до значення `ST0` значення `ST1` і записує результат у `ST0`, а інструкціями

```
fadd   qword [x]
```

неявно завантажує значення x (чотири слова) у сопроцесор, знаходить суму ST0 та x і зберігає результат у ST0.

Аналогічний синтаксис має інструкція для знаходження різниці дійсних чисел fsub. Так інструкція

```
fsub   ST2
```

віднімає від значення ST0 значення ST2 та записує результат віднімання в ST0, а інструкція

```
fsub   qword [y]
```

аналогічно знаходить різницю ST0 та y (чотири слова).

Такий самий синтаксис мають інструкції для знаходження добутку та відношення: fmul та fdiv. Ці інструкції знаходять добуток або відношення ST0 з операндом і зберігають результат у ST0.

Необхідно відзначити, що операнд інструкцій fadd, fdiv, fmul або fsub є коміркою пам'яті, то розмір операнду повинен бути 32 або 64 біти. Операнд, розмір якого 80 біт, повинен бути явно завантажений у сопроцесор інструкцією fld.

Розглянемо програму для обчислення суми *n* членів ряду $S_n = \sum_{i=0}^{n} \frac{1}{2^i}$.

```
1   extern    printf
2   SECTION .data
3         sum  dq   0.0  ; x = 3.14
4         two  dd 1
5         one  dq 1.0
6         cur  dq 0.0
7         n    dd 7
8         frmt db   "sum = %e", 10, 0
9   SECTION .text
10        global main
11  main:
12        mov  ECX, [n]
13        @sum:
14              fild dword [two]     ; ST ← two
15              fld  qword [one]     ; ST ← one
16              fdiv ST1             ; ST ← ST / ST1
17              fadd qword [sum]     ; ST ← ST + sum
18              fstp qword [sum]     ; ST → sum
19              shl  dword [two], 1 ; наступне значення 2^i
20              push ECX        ; зберігаємо значення ECX
21              push dword [sum + 4]
22              push dword [sum]
23              push dword frmt
24              call printf
25              add  ESP, 12
26              pop  ECX        ; відновлюємо значення ECX
```

```
27        loop @sum
28        mov  EAX, 1
29        mov  EBX, 0
30        ret
```

Цикл для обчислення суми знаходиться в рядках 13-27. У рядку 13 поточне значення 2^i завантажується у сопроцесор. У наступному рядку в сопроцесор завантажується значення 1.0. При цьому попереднє значення автоматично проштовхується у регістр `ST1`. Далі у рядку 16 обчислюється відношення значень регістрів `ST0` до `ST1`, результат зберігається у `ST0`. У наступному 17-му рядку до `ST0` додається поточне значення суми. У рядку 18 верхівка стека вивантажується у змінну `sum`, що призводить до оновлення значення суми ряду. Розроблена програма робить такий вивід на екрані:

```
sum = 1.000000e+00
sum = 1.500000e+00
sum = 1.750000e+00
sum = 1.875000e+00
sum = 1.937500e+00
sum = 1.968750e+00
sum = 1.984375e+00
```

Крім наведених вище інструкції сопроцесор підтримує інші[20], частина яких наведена у таблиці 5.

Таблиця 5 – Найбільш поширені інструкції сопроцесора `nasm`

Інструкція	Дія інструкції
`fabs`	Обчислює модуль значення регістра `ST0`, результат зберігається в `ST0`
`fchs`	Зміню знак значення регістра `ST0`: додатне значення замінюється на від'ємне і зворотно
`fcom o`	Порівнює значення `ST0` і `o`, за результатом порівняння змінює відповідні прапорці
`fcos`	Обчислює косинус значення регістра `ST0` (у радіанах), результат зберігається в `ST0`
`ffree fpureg`	Маркує вказаний регістр як пустий
`fpatan`	Обчислює арктангенс (у радіанах) відношення значення регістра `ST1` до `ST0`, результат зберігається в `ST0`
`fptan`	Обчислює тангенс значення регістра (у радіанах) `ST0`,

20 Повний перелік знаходиться за адресою: http://www.csee.umbc.edu/courses/undergraduate/313/fall04/burt_katz/lectures/Lect12/floatingpoint.html

Інструкція	Дія інструкції
	результат зберігається в `ST0`
`fprem`	Обчислює залишок ділення `ST0` на `ST1`, результат зберігається в `ST0`
`frndint`	Округляє значення регістра `ST0` до цілого, результат зберігається в `ST0`
`fsin`	Обчислює синус значення регістра `ST0` (у радіанах), результат зберігається в `ST0`
`fsqrt`	Обчислює квадратний корінь значення регістра `ST0`, результат зберігається в `ST0`

Таким чином арифметичний сопроцесор надає гнучкі можливості створення програм для обробки дійсних чисел. Проте, його практичне використання потребує значною уважності від розробника при програмуванні.

✍ *Змістовна частина завдання*

Індивідуальне завдання ґрунтується на використанні циклічних обчислювальних структур.

Відповідно до варіанту необхідно розробити програму, яка на базі циклів та арифметичного сопроцесора обчислює суму N (повинно бути більшим 10) членів ряду Маклорена функції. Обчислення значення функції необхідно реалізувати окремою підпрограмою.

1. $f(x) = \cos x$;
2. $f(x) = \sin x$;
3. $f(x) = e^x$;
4. $f(x) = \arccos x$;
5. $f(x) = \arcsin x$;
6. $f(x) = \ln(1 + x)$;
7. $f(x) = \operatorname{arctg} x$;
8. $f(x) = \sqrt{1 + x}$;
9. $f(x) = \dfrac{1}{1 - x}$;
10. $f(x) = \operatorname{ch} x$;
11. $f(x) = \operatorname{sh} x$;
12. $f(x) = \operatorname{arch} x$;
13. $f(x) = \operatorname{arsh} x$;
14. $f(x) = \operatorname{arth} x$;
15. $f(x) = \cos 2x$;
16. $f(x) = \sin 2x$;

17. $f(x) = e^{2x}$;
18. $f(x) = \arccos 2x$;
19. $f(x) = \arcsin 2x$;
20. $f(x) = \ln(1 + 2x)$;
21. $f(x) = \operatorname{arctg} 2x$;
22. $f(x) = \sqrt{1 + 2x}$;
23. $f(x) = \frac{1}{1 - 2x}$;
24. $f(x) = \operatorname{ch} 2x$;
25. $f(x) = \operatorname{sh} 2x$;
26. $f(x) = \operatorname{arch} 2x$;
27. $f(x) = \operatorname{arsh} 2x$;
28. $f(x) = \operatorname{arth} 2x$;
29. $f(x) = \cos 3x$;
30. $f(x) = \sin 3x$.

Зробити звіт про виконання індивідуального завдання, який повинен включати постановку завдання, математичну модель розв'язку задачі, блок-схему та набір тестових прикладів зі знімками екрану з роботою програми.

Для нотаток

Завдання для самостійної роботи

Розробка елементарного завантажника операційної системи

Комп'ютерами архітектури x86 для ініціалізації компонентів персональної платформи, необхідних для її первинного завантаження та подальшої роботи, використовується спеціальний набір підпрограм, які об'єднані назвою BIOS[21]. Серед основних завдань, що виконує BIOS, можна виділити:

– знаходить всі підключені пристрої та перевіряє їх справність; якщо найбільш важливі елементи (процесор, пам'ять, відеоадаптер тощо) не знайдені або несправні, то про це сповістить системний динамік, а завантаження операційної системи не відбудеться;

– надає операційній системі базовий набір функцій для взаємодії з обладнанням (наприклад, з екраном та клавіатурою);

– запускає завантажник операційної системи: зчитується завантажувальний сектор – перший сектор носія інформації (порядок опитування носіїв можна змінювати налаштуваннями BIOS).

Розмір початкового завантажника повинен бути всього 512 байт. Цю вимогу було зумовлено розвитком техніки: у дискет дві сторони з кільцевими доріжками (треками), кожен з яких ділиться на дугоподібні частини – сектори в 512 байт. BIOS зчитує перший сектор (завантажувальний) у нульовий сегмент пам'яті за зміщенням 0x7C00, далі за цією адресою передається управління. Початковий завантажник як правило завантажує не саму операційну систему, а іншу програму – більш складний завантажник, що не вміщується в 512 байт.

Для PC-сумісних комп'ютерів завантажувальний сектор повинен містити в двох останніх байтах значення 0x55 і 0xAA – сигнатури завантажувального сектора. Найпростіший завантажник (файл boot.asm) матиме такий вигляд:

```
1   [BITS 16]       ; генерація 16-бітового кода
2   [ORG 0x7C00]    ; Початкове зміщення в пам'яті
3   main:
4         mov  ax, 0x0000
5         mov  ds, ax     ; Ініціалізація сегмента даних
6         mov  si, HelloWorld ; Завантаження рядка
7         call PutStr     ; Виклик процедури друку
8         jmp $           ; Нескінчений цикл
9   ; Процедура друку
10  PutStr:
11        mov ah,0x0E     ; Номер функції BIOS для друку
символу
12        mov bh,0x00     ; Номер сторінки відеопам'яті
13        mov bl,0x07     ; Атрибут тексту
14        .nextchar ; Внутрішня мітка початку циклу
```

21 Basic Input/Output System — базова система введення/виведення

```
15      lodsb     ; інструкція завантаження рядкового
блоку
16      or   al, al    ; Перевірка кінця рядку
17      jz   .return
18
19      int  0x10 ; Переривання BIOS для друку
20      jmp  .nextchar ; Перейти до наступного рядка
21      .return        ; Мітка кінця підпрограми
22      ret       ; Повернення до основної програми
23 ; Дані програми
24 HelloWorld db 'Hello World',13,10,0
25 times 510-($-$$) db 0    ; Заповнення наступних 510
байтів після рядку HelloWorld нулями
26 dw 0xAA55       ; Сигнатура завантажника 511й та 512й
байти
```

Для компіляції програми необхідно виконати команду

```
nasm boot.asm -f bin -o boot.bin
```

Отриманий файл (boot.bin) необхідно записати в завантажувальний сектор образа дискети. Для цього створимо пустий образ дискети

```
dd if=/dev/zero of=boot.img bs=1024 count=1440
```

Далі запишемо в створений образ дискети завантажник

```
dd if=boot.bin of=boot.img conv=notrunc
```

Для тестування створеного образу дискети можна скористатися віртуальною машиною, наприклад, qemu:

```
qemu -fda boot.img -boot a
```

Під час самостійної роботи рекомендується зробити модифікації програми для друку на екрані додаткових повідомлень. Також необхідно продумати організацію взаємодії з додатковими модулями шляхом використання команд умовного та безумовного переходу.

Глосарій

Адреса – число, яке ідентифікує окремі частини пам’яті (комірки) і регістри.

Апаратне забезпечення – комплекс електронних, електричних і механічних пристроїв, що входять до складу системи або мережі.

Архітектура з паралельними процесорами – архітектура обчислювальної системи, що складається з кількох арифметико-логічних пристроїв, які працюють під управлінням одного пристрою керування.

Архітектура комп’ютера – логічна організація, структура та ресурси комп’ютера, які може використовувати програміст. Архітектура визначає принципи дії, інформаційні зв’язки і взаємне з’єднання основних логічних вузлів комп’ютера.

Архітектура подвійної незалежної шини – архітектура устрою процесора, при якому дані передаються по двом шинам незалежно одна від одної, одночасно і паралельно.

Асоціативна пам’ять – в інформатиці – безадресна пам’ять, в якій пошук інформації проводиться за її змістом (асоціативною ознакою).

Багатомашинна обчислювальна система – архітектура обчислювальної системи, що складається їх декількох процесорів, які не мають спільної оперативної пам’яті. Кожен комп’ютер в багатомашинній системі має власну (локальну) пам’ять і класичну архітектуру.

Багатопроцесорна архітектура комп’ютера – архітектура комп’ютера, яка передбачає наявність в комп’ютері декількох процесорів, що дозволяє паралельно обробляти декілька потоків даних і команд.

Базова адреса – початкова адреса, щодо якої ведеться відлік компонентів в масиві даних або машинних команд.

Байт – в запам’ятовувальних пристроях – найменша одиниця даних у пам’яті ЕОМ, яка обробляється як єдине ціле. За замовчуванням байт вважається рівним 8 бітам. Зазвичай в системах кодування даних байт являє собою код одного друкованого або керуючого символу.

Біт – мінімальна одиниця виміру кількості переданої або збереженої інформації, яка відповідає одному двійковому розряду, здатному приймати значення 0 або 1.

Векторний процесор – процесор, що забезпечує паралельне виконання операції над масивами даних (векторами).

Відеоадаптер – електронна плата, яка обробляє відеодані (текст і графіку) і управляє роботою дисплея. Відеоадаптер визначає роздільну здатність дисплея і кількість кольорів.

Відеовведення – пристрій, що забезпечує фізичну взаємодію користувача з персональним комп’ютером протягом роботи з розважальними та діловими програмами і ресурсами Інтернету.

Відеопам’ять – спеціальна пам’ять, яка реалізована на платі управління дисплеєм, призначена для зберігання текстової або графічної інформації, що

відображається на екрані дисплея. Вміст відеопам'яті одночасно доступний процесору і дисплею, що дозволяє змінювати зображення на екрані одночасно з оновленням відеоданих в пам'яті.

Відеосистема комп'ютера – сукупність трьох компонентів: монітора, відеоадаптера і драйверів відеосистеми.

Відкрита архітектура – архітектура комп'ютера або периферійного пристрою, на яку опубліковані специфікації, що дозволяє іншим виробникам розробляти додаткові пристрої до систем з такою архітектурою.

Віртуальний адреса – адреса, яка ідентифікує локалізацію байта у віртуальному адресному просторі. Блок управління пам'яттю транслює віртуальну адресу в фізичну.

Віртуальна пам'ять – ресурси оперативної або зовнішньої пам'яті, що виділяються прикладній програмі операційною системою. Фізичне розташування віртуальної пам'яті на реальних носіях може не збігатися з логічною адресацією даних в прикладній програмі. Перетворення логічних адрес програми у фізичні адреси запам'ятовувальних пристроїв забезпечується апаратними засобами і операційною системою.

Внутрішня пам'ять – пам'ять, що взаємодіє з процесором.

Завдання – в інформатиці – одиниця роботи обчислювальної системи, що вимагає виділення обчислювальних ресурсів: процесорного часу та пам'яті.

Запам'ятовувальний пристрій - пристрій, призначений для зберігання даних.

Запис – структурна одиниця обміну даними між зовнішньою і оперативною пам'яттю.

Захист пам'яті – в мультизадачних обчислювальних системах – особливість управління пам'яттю, яка забороняє яким би то не було процесам доступ до області пам'яті, вже використовуваної іншим процесом. Захист пам'яті забезпечується апаратними засобами і операційною системою.

Зовнішня пам'ять – пам'ять, безпосередньо не доступна центральному процесору. Доступ до зовнішньої пам'яті здійснюється за допомогою обміну даними з оперативною пам'яттю. Зовнішня пам'ять призначена для тривалого зберігання програм і даних. Залежно від запотребованості інформації зовнішня пам'ять підрозділяється на первинну і вторинну.

Інформаційна сумісність – здатність двох або більше ЕОМ або систем адекватно сприймати однаково представлені дані.

Кеш-пам'ять – дуже швидкий запам'ятовувальний пристрій невеликого об'єму, що використовується при обміні даними між процесором і оперативною пам'яттю для компенсації різниці в швидкості обробки інформації процесором і дещо менш швидкодійної оперативної пам'яті.

Кешування – накопичення даних в доступному сховище, з метою їх швидкого вилучення у міру потреби.

Класична архітектура комп'ютера – архітектура комп'ютера, що передбачає один арифметико-логічний пристрій, через який проходить потік даних, і один пристрій управління, через який проходить потік команд.

Команда – опис елементарної операції, яку повинен виконати комп'ютер. Команди зберігаються в комірках пам'яті в двійковому коді.

Комп'ютер – програмований електронний пристрій, здатний обробляти дані і виконувати обчислення, а також інші завдання маніпулювання символами.

Курсор – спеціальна мітка, що показує поточну позицію на екрані.

Логічний запис – сукупність записів взаємопов'язаних елементів даних, яка розглядається в логічному плані як одне ціле. Один логічний запис може складатися з декількох фізичних або бути частиною одного фізичного запису.

Масштабована процесорна архітектура – розроблена корпорацією Sun 32-розрядна архітектура родини чипів, яка базується на концепції обчислень зі скороченим набором команд (RISC).

Математичний сопроцесор – сопроцесор, який виконує операції над числами, представленими у формі з плавучою точкою.

Матричний процесор – процесор, який має архітектуру, розраховану на обробку числових масивів (матриць). Архітектура процесора містить у собі матрицю процесорних елементів, що працюють одночасно.

Мейнфрейм – високопродуктивний комп'ютер загального призначення зі значним обсягом оперативної і зовнішньої пам'яті, призначений для виконання інтенсивних обчислювальних робіт. Зазвичай з мейнфреймом працює множина користувачів, кожен з яких має лише термінал, позбавлений власних обчислювальних потужностей.

Мікропроцесор – процесор, виконаний у вигляді однієї або декількох взаємозалежних інтегральних схем. Мікропроцесор складається з ланцюгів керування, регістрів, суматорів, лічильників команд і дуже швидкої пам'яті малого об'єму.

Монітор – пристрій візуального відображення інформації у вигляді тексту, таблиць, малюнків, креслень тощо.

Мультипроцесор – комп'ютер, що має кілька процесорів і працює в режимі багатопроцесування.

Надлишковий масив незалежних дисків (RAID) – архітектура масиву жорстких дисків, що забезпечує відмовостійкість накопичувачів. Рівні специфікації розрізняються за продуктивністю, надійністю та ціною.

Пам'ять – в інформатиці – здатність об'єкта забезпечувати зберігання даних. Зберігання здійснюється в запам'ятовувальних пристроях.

Персональний комп'ютер – універсальна ЕОМ, призначена для індивідуального використання. Зазвичай персональні комп'ютери проектуються на основі принципу відкритої архітектури і створюються на базі мікропроцесорів.

Повна сумісність – технічна, програмна та інформаційна сумісність двох або більше ЕОМ без будь-яких обмежень для їх користувачів.

Послідовний комп'ютер – комп'ютер з жорсткою послідовністю операцій, в якому в кожний момент часу тільки один процесорний елемент обробляє одну команду. Послідовний комп'ютер реалізує архітектуру обчислень Фон-Неймана.

Програма – згідно з ГОСТом 19781-90 – дані, призначені для управління конкретними компонентами системи обробки інформації з метою реалізації певного алгоритму.

Програмна сумісність – можливість виконання одних і тих же програм на різних ЕОМ з отриманням однакових результатів.

Процесор – пристрій, призначений для виконання команд.

Процесор зображень – векторний процесор, призначений для обробки сигналів, що надходять від датчиків-формувачів зображення.

Процесор MISC – процесор, що працює з мінімальним набором довгих команд, упакованих в одне слово розміром 128 біт.

Процесор RISC – процесор, в якому реалізовано спрощений набір команд, які мають однакову довжину; більшість команд виконуються за один цикл процесора, відсутні макрокоманди, взаємодія з оперативною пам'яттю обмежено операціями пересилання даних, реалізований мінімальний набір способів адресації пам'яті, реалізований конвеєр команд, використовується високошвидкісна пам'ять.

Процесор VLIW – процесор, що працює з системою команд надвеликої розрядності.

Прямий доступ до пам'яті – метод звернення зовнішнього пристрою до пам'яті комп'ютера без участі центрального процесора.

Розподіл пам'яті – управління ресурсами пам'яті в інтересах вирішення окремих завдань.

Скалярний процесор – процесор з традиційною фон-неймановою архітектурою, що оперує тільки зі скалярними даними.

Сопроцесор – допоміжний процесор, призначений для виконання математичних і логічних дій, які не входять в стандартний набір команд центрального процесора. Використання сопроцесора дозволяє прискорити процес обробки інформації комп'ютером.

Структура комп'ютера – сукупність функціональних елементів комп'ютера і зв'язків між ними. Графічно структура комп'ютера представляється у вигляді схем, за допомогою яких можна отримати опис комп'ютера на будь-якому рівні деталізації.

Сумісність ЕОМ – здатність одного комп'ютера сприймати і обробляти дані так само, як інший комп'ютер, без модифікації даних або носія, за допомогою якого вони передаються.

Суперкомп'ютер – потужний комп'ютер з продуктивністю понад 100 мільйонів операцій з плавучою точкою в секунду. Суперкомп'ютер являє собою багатопроцесорний та / або багатомашинний комплекс, що працює на загальну пам'ять і загальне поле зовнішніх пристроїв.

Технічна сумісність – здатність однієї ЕОМ працювати з вузлами або пристроями, що входять до складу іншої ЕОМ.

Файл – сукупність пов'язаних записів (кластерів), які зберігаються у зовнішній пам'яті комп'ютера і розглядаються як єдине ціле. Зазвичай файл однозначно ідентифікується зазначенням імені файлу, його розширенням і

шляхом доступу до файлу. Кожен файл складається з атрибутів і вмісту. Розрізняють текстові, графічні, звукові та інші файли.

Фізичний запис – порція даних, яка пересилається як одне ціле між оперативною і зовнішньою пам'яттю ЕОМ.

Шина – фізичний засіб, до якого однаковим чином підключається група взаємодійних один з одним комп'ютерів або їх компонентів. Для створення шин використовуються плоскі кабелі. Сукупність проводів шини розділяється на окремі групи: шину адреси, шину даних і шину управління. Розрізняють також системні та локальні шини.

Complex Instruction Set Chip (CISC) – процесор з повним набором команд, що виконує до 300 машинних інструкцій.

Рекомендована література

Основна:

1. Абель П. Ассемблер. Язык и программирование для IBM PC / Питер Абель. – М. : Корона-Век, 2007. – 736 с.

2. Марек Р. Ассемблер на примерах. Базовый курс / Рудольф Марек. – СПб. : Наука и Техника, 2005. – 240 с.

3. Поворознюк А.И. Архитектура компьютеров. Архитектура микропроцессорного ядра и системных устройств. Ч.1 / А.И. Поворознюк. – Х. : Торнадо, 2004. – 355 с.

4. Поворознюк А.И. Архитектура компьютеров. Архитектура микропроцессорного ядра и системных устройств. Ч.2 / А.И. Поворознюк. – Х. : Торнадо, 2004. – 296 с.

5. Рудаков П.И. Язык Ассемблера: уроки программирования / П.И. Рудаков, К.Г. Финогенов. – М. : Диалог-мифи, 2001. – 640 с.

6. Таненбаум Э. Архитектура компьютера / Э. Таненбаум. – СПб. : Питер, 2007. – 844 с.

7. Юров В.И. Assembler. Учебник для вузов / В.И. Юров. – СПб. : Питер, 2003. – 637 с.

8. Столяров А.В. Программирование на языке ассемблера NASM для ОС Unix / А.В. Столяров. – М. : МАКС Пресс, 2011. – 188 с.

Додаткова:

1. Баула Б.Г. Архитектура ЭВМ и операционные среды / В.Г. Баула, А.Н. Томилин, Д.Ю. Волканов. – М. : Академия, 2011. – 336 с.

2. Королев Л.Н. Архитектура ЭВМ / Л.Н. Королев. – М. : Научный мир, 2005. – 272 с.

3. Максимов Н.В. Архитектура ЭВМ и вычислительных систем / Н.В. Максимов, И.И. Попов, Т.Л. Партыка. – М. : Форум, 2010. – 512 с.

4. Краковяк С. Основы организации и функционирования ОС ЭВМ / С. Краковяк. – М. : Мир, 1988. – 480 с.

5. Лехин С.Н. Схемотехника ЭВМ / С.Н. Лехин. – СПб. : БХВ-Петербург, 2010. – 672 с.

6. Орлов С.А. Организация ЭВМ и систем / С.А. Орлов, Б.Я. Цилькер. – СПб. : Питер, 2011. – 688 с.

7. Горнец Н.Н. Организация ЭВМ и систем / Н.Н. Горнец, А.Г. Рощин, В.В. Соломенцев. – М. : Академия, 2008. – 320 с.

8. Скэнлон Л. Персональные ЭВМ IBM PC и XT. Программирование на языке ассемблера / Л. Скэнлон. – М. : Радио и связь, 1989. – 336 с.

Інформаційні ресурси:

1. http://asmworld.ru/
2. http://www.stolyarov.info/books/pdf/nasm_unix.pdf

3. http://www.codenet.ru/progr/asm/nasm/
4. http://citforum.ru/programming/tasm3/index.shtml
5. http://www.nasm.us/xdoc/2.11.01/html/nasmdoc0.html
6. http://www.csee.umbc.edu/courses/undergraduate/313/fall04/burt_katz/lectures/Lect12/index.html
7. http://wiki.osdev.org/Main_Page
8. http://osdev.ru/
9. http://www.osdever.net/tutorials/
10. http://the-programmer.ru/load/video_uroki_assembler/assembler_video_uroki_dlja_nachinajushhikh/55

Додаток А. Приклад консольної програми для операційної системи Windows

Нижче наведено приклад консольної програми для операційної системи Windows, реалізованої на базі діалекту асемблера NASM.

```
1   ;*************************************************
*****
2   ;* Windows console code to write to stdout
3   ;*************************************************
*****
4   ;Nasm win32 code
5   ;Minimalistic code to write to stdout under Microsoft
Windows
6   ;compile with:
7   ;nasm -fobj winstdout.nasm -l winstdout.lst
8   ;link with:
9   ;alink -c -oPE -subsys console winstdout
10          [list -]
11          %include "win32n.inc"
12          [list +]
13  section .bss    use32
14  section .data   use32
15  section .code   use32
16          cpu     386
17  ;External functions
18          extern  ExitProcess
19          import  ExitProcess kernel32.dll
20          extern  GetStdHandle
21          import  GetStdHandle kernel32.dll
22          extern  WriteFile
23          import  WriteFile kernel32.dll
24  ;*************************************************
*****
25  ;* Executable code start
26  ;*************************************************
*****
27  section .code
28  ..start:
29  ;Get a handle to stdout
30          push    dword STD_OUTPUT_HANDLE
31          call    [GetStdHandle]          ;Get stdout
32          cmp     eax, -1                 ;Not found?
33          je      err_exit
```

```
34          mov     [stdout_handle], eax    ;Save the
handle
35  ;Write a message to stdout
36          push    dword 0                 ;inout
lpOverlapped
37          push    dword octets_written    ;out
38          push    dword hello_text_len    ;in octets to
write
39          push    dword hello_text        ;in buffer
40          push    dword [stdout_handle]   ;in handle
41          call    [WriteFile]             ;Write
message
42          or      eax, eax                ;Zero
indicates error
43          jz      err_exit
44
45  exit:
46          push    dword error_code        ;Point at
error code
47          call    [ExitProcess]
48          jmp     exit                    ;Should never
reach here
49  err_exit:
50  ;Since we cannot write to a standard file descriptor
there's not
51  ;much we can to to explain what went wrong. We can
return an
52  ;error code, however. In real code you could try
popping up a
53  ;message box.
54          mov     eax, 1
55          mov     [error_code], eax
56          jmp     exit
57  ;*****************************************************
*****
58  ;* Initialised data
59  ;*****************************************************
*****
60  section .data
61  error_code:     dd      0 ;Default return code
62  stdout_handle:  dd      0
63  octets_written: dd      0 ;Return from WriteFile call
64  hello_text:     db      "Hello", 13, 10
65  hello_text_len: equ     $ - hello_text
```

Додаток Б. Арифметичні операції на базі системи з архітектурою x32

Наведений нижче програмний є реалізацією прикладу з лабораторної роботи № 6 для 32-бітних систем.

```
1   extern    printf
2   SECTION .data        ; Секція даних
3        a    dd   103   ; int a = 103;
4        b    dd   259   ; int b = 257;
5        c    dd   0     ; int c = 0;
6        fmt  db   "a=%d, b=%d, c=%d", 10, 0 ; Рядок
формату функції printf, "\n",'0'
7        fmt_add   db   "c = a + b = %d", 10, 0   ; Рядок
формату для друку суми
8        fmt_sub   db   "c = a - b = %d", 10, 0   ; Рядок
формату для друку різниці
9        fmt_mul   db   "c = a * b = %d", 10, 0   ; Рядок
формату для друку добутку
10       fmt_div   db   "[a / b] = %d, {a / b} = %d", 10,
0   ; Рядок формату друку для знаходження частки а
залишку
11  SECTION .text             ; Секція коду
12       global main          ; Стандартна точка старту
gcc
13  main:                     ; мітка для точки старту
14       push dword     [c]
15       push dword     [b]
16       push dword     [a]
17       push dword     fmt
18       call printf          ; Виклик функції для друку
значень a, b, c
19       add  ESP, 16
20                      ; Обчислення c = a + b
21       mov  EAX, [a]  ; EAX = a
22       add  EAX, [b]  ; EAX = EAX + b = a + b
23       mov  [c], EAX  ; c = EAX = a + b
24                      ; Друк
25       push dword[c]
26       push dword     fmt_add
27       call printf    ; Виклик функції для друку
28       add  ESP, 8
29                      ; Обчислення c = b - a
30       mov  EAX, [b]  ; EAX = b
31       sub  EAX, [a]  ; EAX = EAX - a = b - a
```

```
32      mov  [c], EAX  ; c = EAX = b - a
33                     ; Друк
34      push dword     [c]
35      push dword     fmt_sub
36      call printf          ; Виклик функції для друку
37      add  ESP, 8
38                           ; Обчислення c = a * b
39      mov  EAX, [a]        ; EAX = a
40      mul  dword     [b]   ; EDX:EAX = EAX * b = a * b
41      mov  [c], EAX        ; c = EAX = a * b
42                           ; Друк
43      push dword     [c]
44      push dword     fmt_mul
45      call printf          ; Виклик функції для друку
46      add  ESP, 8
47                           ; Обчислення c = b / a
48      mov  EAX, [b]        ; EAX = b
49      mov  EDX, 0
50      div  dword     [a]   ; EAX - частка, EDX -
залишок
51                           ; Друк
52      push dword     EDX
53      push dword     EAX
54      push dword     fmt_div
55      call printf          ; Виклик функції для друку
56      add  ESP, 12
57      mov  EAX, 0
58      ret
```

Додаток В. Текст програми з синтаксисом `Intel`

Наведена програма є результатом трансляції програми з лабораторної роботи №12 компілятором `gcc`. Необхідно звернути увагу на виклик організацію підпрограми в рядках 6-32 та її виклику в рядках 50-51. Наведений код є результатом трансляції на машині з 64-бітною архітектурою.

```
1       .file"asm_inline.c"
2       .intel_syntax noprefix
3       .text
4       .globl    factorial
5       .typefactorial, @function
6   factorial:
7   .LFB0:
8       .cfi_startproc
9       push rbp
10      .cfi_def_cfa_offset 16
11      .cfi_offset 6, -16
12      mov  rbp, rsp
13      .cfi_def_cfa_register 6
14      mov  DWORD PTR [rbp-20], edi
15      mov  edx, DWORD PTR [rbp-20]
16  #APP
17  # 7 "asm_inline.c" 1
18      .intel_syntax noprefix
19      mov ecx, edx
20      mov eax, 1
21      0:
22      mul ecx
23      loop 0b
24      mov edx, eax
25
26  # 0 "" 2
27  #NO_APP
28      mov  DWORD PTR [rbp-4], edx
29      mov  eax, DWORD PTR [rbp-4]
30      pop  rbp
31      .cfi_def_cfa 7, 8
32      ret
33      .cfi_endproc
34  .LFE0:
35      .sizefactorial, .-factorial
36      .section  .rodata
37  .LC0:
38      .string   "f = %d\n"
```

```
39      .text
40      .globl    main
41      .typemain, @function
42  main:
43  .LFB1:
44      .cfi_startproc
45      push rbp
46      .cfi_def_cfa_offset 16
47      .cfi_offset 6, -16
48      mov  rbp, rsp
49      .cfi_def_cfa_register 6
50      mov  edi, 3
51      call factorial
52      mov  esi, eax
53      mov  edi, OFFSET FLAT:.LC0
54      mov  eax, 0
55      call printf
56      mov  eax, 0
57      pop  rbp
58      .cfi_def_cfa 7, 8
59      ret
60      .cfi_endproc
61  .LFE1:
62      .sizemain, .-main
63      .ident    "GCC: (Ubuntu/Linaro 4.8.1-10ubuntu9)
4.8.1"
64      .section  .note.GNU-stack,"",@progbits
```

Для нотаток

Методичне видання
(українською мовою)

Чопоров Сергій Вікторович

ОРГАНІЗАЦІЯ І ФУНКЦІОНУВАННЯ ЕОМ

Навчально-методичний посібник до лабораторних занять
для студентів освітньо-кваліфікаційного рівня «бакалавр»
напряму підготовки «Програмна інженерія»

Рецензент *С.Ю. Борю*
Відповідальний за випуск *С.І. Гоменюк*
Коректор *Н.В. Плюта*